Event-Driven Programming Techniques

This chapter addresses the more experienced GUI programmer and describes essential programming techniques. There are two ways to program in the dialog editor:

- Use the dialog editor’s menu bar and toolbar to create new dialogs or dialog elements and use the attributes window to assign attribute values to them. The dialog editor will internally generate the corresponding Natural code.
- Open an event-handler section or an inline-subroutine section and specify Natural code explicitly. This code will be added to the code that is generated internally. You can also enter parameter data areas, global data areas and local data areas in the corresponding definition sections.

You can view the current dialog’s generated and specified code by choosing "Object > List" in the dialog editor’s menu bar.

If you want a hands-on demonstration of how to program with the dialog editor, refer to the SYSEXEVT library. This library contains sample dialogs demonstrating basic functionality. Before accessing the sample dialogs, read the README file. Then execute the MENU dialog.

Notes:
Code written in the dialog editor must be in structured mode.

If you want to execute a Natural application using dialogs, you must use a dialog to start this application.

The following topics are covered below:

- Working with Status Bar Controls
- Working with Dynamic Information Line and Status Bar
- Adding a Maximize/Minimize/System Button
- Defining Color
- Adding Text in a Certain Font
- Adding Online Help
- Defining Mnemonic and Accelerator Keys
- Dynamic Data Exchange - DDE
- Object Linking and Embedding - OLE

For further information on Event-driven Programming see Introduction to Event-Driven Programming.

How To Open and Close Dialogs

Opening a Dialog

An event-driven application is started by executing the base dialog. Events triggered by the end user will then typically cause other dialogs to be started. The application ends when the base dialog is closed.

To open a dialog from anywhere within an event-driven application

Use the statement OPEN DIALOG.

This statement causes the dialog to be loaded and the processing on its opening to be performed.

Control over processing returns from the opened dialog except for dialogs with the style "Dialog Box". For those dialog styles, control returns only after the dialog has ended.

The parameters passed are accessible only during the processing on the opening of a dialog (before-open and after-open events), except for when the parameters are declared as BY VALUE in the parameter data area of the opened dialog or when the dialog has the style "Dialog Box".

To open a dialog from anywhere within an event-driven Natural application, the following syntax is used:

```
OPEN DIALOG operand1 [USING] [PARENT] operand2
                     [[GIVING] [DIALOG-ID] operand3]
                     [     { operand4...         } ]
                     [ WITH{                     } ]
                     [     { PARAMETERS-clause   } ]
```

Operands

Operand1 is the name of the dialog to be opened. If the *PARAMETERS-clause* is used, *operand1* must be a constant (the name of a cataloged dialog).

Operand2 is the handle name of the parent.

Operand3 is a unique dialog ID returned from the creation of the dialog. It must be defined with format/length I4.

Passing Parameters to the Dialog

When a dialog is opened, parameters may be passed to this dialog.

As *operand4* you specify the parameters that are passed to the dialog.

With the *PARAMETERS-clause*, parameters may be passed selectively:

```
PARAMETERS {parameter-name =operand4}... END-PARAMETERS
```

Note: You may only use the PARAMETERS-clause if operand1 is an alphanumeric constant and if the dialog is cataloged.

Parameter-name is the name of the parameter as defined in the parameter data area section of the dialog.

To avoid format/length conflicts between operands and parameters passed, see the BY VALUE option of the DEFINE DATA statement in the Natural Statements Manual.

When passing parameters only with *operand4*, a dialog may be opened as follows:

Example:

```
/* The following parameters are defined in the calling dialog's parameter
/* data area (not in the parameter data area of the dialog to be opened):
1 #MYDIALOG-ID (I4)
1 #MYPARM1  (A10)
/* Pass the operands #MYPARM1 and 'MYPARM2' to the parameters #DLG-PARM1 and
/* #DLG-PARM2 defined in the dialog to be opened:
OPEN DIALOG 'MYDIALOG'
  USING #DLG$WINDOW
  GIVING #MYDIALOG-ID
WITH #MYPARM1 'MYPARM2'
```

When passing parameters selectively with the *PARAMETERS-clause*, a dialog may be opened as shown in the following example:

Example:

```
/* The following parameters are defined in the calling dialog's parameter
/* data area (not in the parameter data area of the dialog to be opened):
1 #MYDIALOG-ID (I4)
1 #MYPARM1 (A10)
/* Pass the operands #MYPARM1 and 'MYPARM2' to the parameters #DLG-PARM1 and
/* #DLG-PARM2 defined in the dialog to be opened:
OPEN DIALOG 'MYDIALOG'
  USING #DLG$WINDOW
  GIVING #MYDIALOG-ID
WITH PARAMETERS
   #DLG-PARM1=#MYPARM1
   #DLG-PARM2='MYPARM2'
END-PARAMETERS
```

Permanence In Creating, Passing And Checking Data

The term "permanence" is used in Natural to denote data defined in a base dialog's local data area whose existence is guaranteed throughout the whole lifetime of the dialog. Data defined in the global data area are not kept permanent because the global data area can be exchanged while the application is executed.

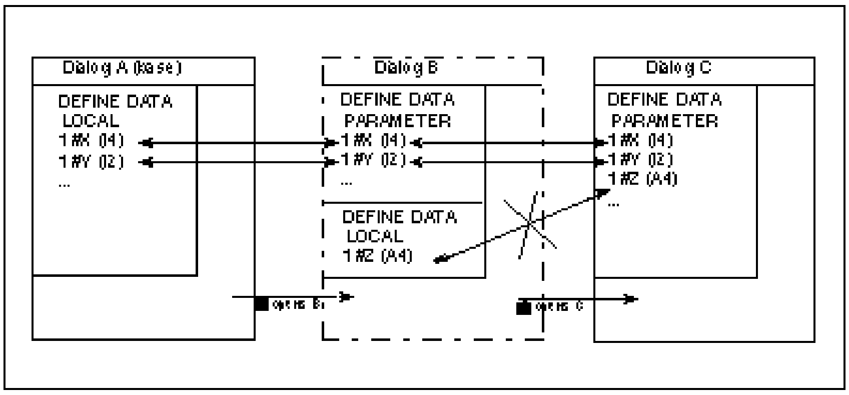
The reference to the permanent data is kept by saving the parameter data area internally during opening of the dialog. This reference is reused when

- a dialog element receives an event;
- all parameters passed from one dialog to another are permanent, provided they reference the base dialog's local data area.

Parameters are accessible

- during the before-open and after-open event processing on opening of a dialog or
- if *all of them* reference the base dialog's local data area.

The following example illustrates a case in which two parameters are kept permanently and one other is not. Assume the base dialog is dialog A. This base dialog now opens dialog B, passing parameters #X and #Y. After that, dialog B passes parameters #X and #Y on to dialog C. The #X and #Y parameters which are now in dialog C will be permanent, even if dialog B is closed. If, however, dialog B passes its own parameter #Z when opening dialog C, the parameter #Z is not permanent, because if dialog B is closed, the reference to its local data area is no longer valid. No parameter in dialog C is accessible (#Z does not reference the base dialog's local data area).



Processing Steps When Opening a Dialog

This section describes what happens when a dialog is opening. You can open a dialog either by executing it, for example from the command line, or by invoking it with an OPEN DIALOG statement.

- The dialog object is loaded and starts executing.
- The BEFORE-ANY event-handler section is executed, the value of the system variable *EVENT being OPEN.
- The BEFORE-OPEN event-handler section is executed.
- The dialog window is created as specified in the dialog editor.
- The BEFORE-ANY event-handler section is executed. *EVENT = AFTER-OPEN.
- All dialog elements are created as specified in the dialog editor.
- The dialog window and all dialogs are made visible except those that are VISIBLE = FALSE.
- The AFTER-OPEN event-handler section is executed.
- The AFTER-ANY event-handler section is executed. *EVENT = AFTER-OPEN.
- The AFTER-ANY event-handler section is executed. *EVENT = OPEN (not if the dialog’s STYLE attribute value is "Dialog Box").
-

Closing Dialogs

To close a dialog dynamically, you specify the following:

```
CLOSE DIALOG [USING] [DIALOG-ID] { operand1   }
                                 { *DIALOG-ID }
```

Operand1 is the identifier of the dialog as returned in the OPEN DIALOG statement.

Example:

```
CLOSE DIALOG *DIALOG-ID /* Close the current Dialog
```

The dialog will then be erased from the screen and removed from memory. All local data associated with the dialog will be gone.

Note: If a modal dialog is a child in a hierarchy of dialogs, the modal dialog should not close its parent(s) because this will result in a deadlock.

operand1

Operand1 is the name of the dialog to be closed.

To close the current dialog, you specify *DIALOG-ID.

Initializing Attribute Values

You can specify conditions for the opening and closing of a dialog: this applies to the before-open, after-open, and close events. These conditions can be used to initialize the attribute values in the dialog.

The following is an example of after-open event-handler code: Red foreground color is assigned to push buttons that the user must press after entering data in the associated input fields.

Example:

```
DEFINE DATA LOCAL
  ...
  1 #OK-BUTTON HANDLE OF PUSHBUTTON
  1 #CALC-BUTTON HANDLE OF PUSHBUTTON
  1 #SAVE-BUTTON HANDLE OF PUSHBUTTON
  1 #CONVERT-BUTTON HANDLE OF PUSHBUTTON
  ...
END-DEFINE
...
#OK-BUTTON.FOREGROUND-COLOUR-NAME := RED
#CALC-BUTTON.FOREGROUND-COLOUR-NAME := RED
#SAVE-BUTTON.FOREGROUND-COLOUR-NAME := RED
#CONVERT-BUTTON.FOREGROUND-COLOUR-NAME := RED
```

If you want to modify attribute values of dialog elements and of the dialog before the dialog is opened (and displayed to the end user), do not specify this in the "before open" event-handler code, because the dialog elements and the dialog window are not yet created. Instead, create the dialog with the dialog editor and set the attribute VISIBLE to FALSE in the "Dialog Attributes" window. Then modify all the attribute values in the after-open event-handler code (when the handles are available). Then make the dialog visible with VISIBLE = TRUE.

Example:

```
DEFINE DATA LOCAL
  ...
  1 #DIA-1 HANDLE OF DIALOG
  1 #OK-BUTTON HANDLE OF PUSHBUTTON
  1 #CALC-BUTTON HANDLE OF PUSHBUTTON
  ...
END-DEFINE
...
/* AFTER OPEN event-handler code section
...
#OK-BUTTON.FOREGROUND-COLOUR-NAME := RED
#CALC-BUTTON.FOREGROUND-COLOUR-NAME := RED
#DIA-1.VISIBLE := TRUE
```

How To Edit a Dialog’s Enhanced Source Code

What Is The Enhanced Source Code Format ?

The enhanced source code format enables you to edit source code that has been generated by the dialog editor. You edit enhanced source code in a program editor window. When you edit a dialog, the dialog editor stores the results in internal structures. From these structures, source code is generated when you save, stow, list or execute any other system command on the dialog. Code is also generated when you refresh the program editor’s source code window.

You can edit enhanced source code as you do any other Natural user code. The source code syntax is subject to a number of formal conventions, however. For a documentation of the enhanced source code syntax, see The Enhanced Source Code Format in the Dialog Components Manual.

When you execute a system command on a dialog you have just edited in the program editor source code window, the dialog editor updates its internal structures and refreshes the source code window.

Note: The dialog editor preserves code layout only in the user code sections, such as event handlers.

The dialog editor supports the following source formats:

- 213. This is the format generated by Natural Version 2.1.3 (New Dimension). It is supported for input only. You cannot generate 2.1.3 format with Natural Version 3.1 and Version 3.2.
- 22C. This is the format generated by Natural Version 2.2.2. In Natural for Windows and Unix/OpenVMS Version 4.1, dialogs can no longer be generated in this format. It, too, is supported for input only.
- 22D. This is the "enhanced" source-code format that from now on is the standard. It is generated for compiling, storing, and editing dialogs in Natural Version 2.2.3 and above.

The characteristics of the enhanced source code format are:

- Dialog sources are readable and printable without requiring conversion.
- Dialog sources consist only of legal and fully documented Natural syntax.
- Dialog sources can be edited textually using program editor functions such as scanning for and replacing text.
- Dialog sources can be displayed in the Natural Debugger.
- Dialog sources are larger than 213 or 22C format sources (by a factor between 1.25 and 3.5).
- Any code that can be generated with the dialog editor can also be coded manually. For example, if you "draw" a push-button control onto the user interface, the corresponding code is generated implicitly. You can also create this push-button control explicitly with the help of a source-code window that provides you with the functions of the program editor.
- You can switch between the dialog editor and the program editor by selecting the source code window or the dialog window. If you edit in either window, you need to synchronize your updates: (graphically) modifying the dialog locks the source code window and you may not make changes there. Correspondingly, if you change the source code, you may not make changes in the dialog window, which is locked. If your editor is locked, its status bar displays "Locked".

For dialogs in the old formats, this means:

- They remain unchanged until they are processed in the dialog editor. They can be compiled and executed in their old format.
- When you load them into the dialog editor, the dialogs are saved in the new format. If they are saved in the enhanced format, you must include the local data area NGULKEY1. Note that the storage size increases when the dialogs are saved.
- When you list or print them and you enable the "enhanced list mode" option, the dialogs are displayed using the enhanced source code format.

Avoiding Incompatibilities Between Dialog Editor And Program Editor

When you edit the enhanced source code format, note that some of the syntax elements accepted by the program editor are not accepted by the dialog editor. Enhanced source code editing is not intended as a new programming technique in addition to using the dialog editor:

- It may be syntactically acceptable to replace a dialog element’s numeric coordinate (a RECTANGLE-X attribute value) with a variable reference. The dialog editor, however, will not accept this when the changes are synchronized, and will prompt you when you issue a command requiring the source code.
- The dialog editor may accept a reference to a variable’s STRING attribute even if the variable is not declared, but the compiler will not accept this.

In the sections that are not user code, you should avoid such incompatibilities by adding only code that is acceptable to both the compiler and the dialog editor.

In the user code sections, such as in event-handler sections and in external or internal subroutines, your choice of programming techniques is not restricted by the dialog editor. In these sections, however, you have no visual editing support.

As a general rule, a mixed approach is often the best, especially when you use dialog-editor- generated code as a starting point.

Note: In the dialog editor, you can copy dialog elements to the clipboard and when you paste them into user code, they appear as text.

How To Use The Enhanced Source Code Format

To edit a dialog in the enhanced source code format

1. Load the dialog into the dialog editor.
2. From the "Dialog" menu, choose "Source Code".
   Or choose the "Source Code" toolbar button.
   Or press CTRL+ALT+C.

The dialog’s source code window appears and the program editor is loaded. This editor enables you to scan for text strings, replace them, and so on. For more information on how to use the program editor, see The Program Editor.

The enhanced source code format’s syntactical conventions are documented in the chapter The Enhanced Source Code Format in the Dialog Components Manual.

Enhanced source code can be listed and printed as usual. You can also scan for strings by using the Find option of the Edit menu.

Note: If you are replacing strings with this option, this can make a dialog source incompatible with the dialog editor.

How Dialogs, Controls and Items Are Related Hierarchically

Dialogs and their dialog elements are organized hierarchically. Typically, the dialog window contains a number of controls. The controls are children of the window or of other controls which are capable of acting as containers. A control may contain a number of items. For example, a list-box control may contain several list-box items. The control is the parent of the items.

The dialogs themselves are also organized hierarchically. Every time the OPEN DIALOG statement is specified, the parent of the newly created dialog must be provided as a parameter. This parameter may be NULL-HANDLE or the handle of an existing dialog. If NULL-HANDLE is provided, the dialog belongs to the desktop rather than to any other dialog. This means that the dialog can be closed and minimized independently of any other dialog in the application. A dialog having an existing dialog as parent is closed or minimized when the parent dialog is closed or minimized.

The first dialog in an application plays a special role and is sometimes called the base dialog. When the base dialog is closed, all other dialogs in the application are also closed, whether they are children of the base dialog or not.

All children on one hierarchical level are sorted in the sequence of their creation. Each dialog element therefore always "knows" its parent, its predecessor and successor (on the same hierarchical level), and its first and last child (if present). You can retrieve this information by using the following attributes:

- PARENT
- PREDECESSOR
- SUCCESSOR
- FIRST-CHILD
- LAST-CHILD

These attributes contain handle values of dialog elements. If their value is NULL, the dialog element has no parent, successor, or child. The following example demonstrates how to go through all dialog elements of a dialog.

Example 1:

```
1 #CONTROL HANDLE OF GUI

#CONTROL := #DLG$WINDOW.FIRST-CHILD
REPEAT UNTIL #CONTROL = NULL-HANDLE
  ...
  #CONTROL := #CONTROL.SUCCESSOR
END-REPEAT
```

List-box controls and list-box items contain an additional attribute:

SELECTED-SUCCESSOR can be set for either the list-box control itself or for any of its items. It points to the next selected item in a list-box control. For the list-box control itself, it points to the first selected item.

Example 2:

```
1 #ITEM HANDLE OF LISTBOXITEM

#ITEM := #LISTBOX.SELECTED-SUCCESSOR
REPEAT UNTIL #ITEM = NULL-HANDLE
  ...
  #ITEM := #ITEM.SELECTED-SUCCESSOR
END-REPEAT
```

The above example is the query necessary to find all selected items in a list-box control where multiple selection is allowed (MULTI-SELECTION attribute).

How To Define Dialog Elements

Dialog elements are uniquely identified by a handle. A handle is a binary value that is returned when a dialog element is created. A handle must be defined in a DEFINE DATA statement of the dialog.

You can define a handle

- by creating a dialog or a dialog element with the dialog editor; in this case, the handle definition is generated;
- by explicitly entering the definition in a global, local, or parameter data area of the dialog;
- by explicitly entering the definition in a subprogram or a subroutine.

Note: Handles of ActiveX controls are defined in a slightly different way than the standard handle definition described below. This is described in Working with ActiveX Controls.

A handle is defined inside a DEFINE DATA statement in the following way:

```
level handle-name [(array-definition)] HANDLE OF dialog-element-type
```

Handles may be defined on any *level*.

Handle-name is the name to be assigned to the handle; the naming conventions for user-defined variables apply.

Dialog-element-type is the type of dialog element. Its possible values are the values of the TYPE attribute. It may not be redefined and not be contained in a redefinition of a group.

Examples:

```
1 #SAVEAS-MENUITEM HANDLE OF MENUITEM
1 #OK-BUTTON (1:10) HANDLE OF PUSHBUTTON
```

When you have defined a handle, you can use the *handle-name* with handle attribute operands in those Natural statements where an operand may be specified. With handle attribute operands, you can, for example, dynamically query, set, or modify attribute values for the defined *dialog-element-type*. This is the most important programming technique in the dialog editor. For details, see the section How To Manipulate Dialog Elements.

If there is a dialog element handle of the same name in two different dialogs, the PARENT attribute ensures that Natural knows the difference between the two handles (two different PARENT values). Handles may be passed as parameters or may be assigned from one handle variable to another.

HANDLE OF GUI

In addition to the handle types referring to one dialog element, the generic handle type HANDLE OF GUI is available. In event-handler code, you can use HANDLE OF GUI to refer to the handle of any type of dialog element.

This can be useful, for example, if you are querying an attribute value in all dialog elements on one level: you go through the dialog elements one after the other; in the course of this query, it is not clear which type of dialog element is going to be queried next. Then a GUI handle makes it possible to query the next dialog element regardless of its type. This saves a lot of coding, because otherwise, you would have to query the attribute's value of each dialog element separately.

Example:

```
...
1 #CONTROL HANDLE OF GUI
...
#CONTROL := #DLG$WINDOW.FIRST-CHILD
REPEAT UNTIL #CONTROL = NULL-HANDLE
  ...
  #CONTROL := #CONTROL.SUCCESSOR
END-REPEAT
```

NULL-HANDLE

The HANDLE constant "NULL-HANDLE" may be used to query, set or modify a NULL value of a HANDLE. Such a NULL value means that the dialog element is nonexistent (even if it has been created explicitly).

Example:

```
DEFINE DATA PARAMETER
  1 #PUSH HANDLE OF PUSHBUTTON
END-DEFINE
...
IF #PUSH = NULL-HANDLE
...
```

The HANDLE constant "NULL-HANDLE" represents the NULL value of a HANDLE variable or of an attribute with format HANDLE. For handle variables, the value indicates that the expression *handle.attribute* refers to the global attribute list. For attributes, this value indicates that no value is currently set.

How To Manipulate Dialog Elements

To manipulate dialog elements, Natural provides you with handle attribute operands. You use handle attribute operands wherever an operand may be specified in a Natural statement. This is the most important programming technique in event-handler code.

Important: You must have defined a handle.

Note: ActiveX controls are manipulated in a slightly different way than the standard way described below. This is described in Working with ActiveX Controls.

Handle attribute operands may be specified as follows:

```
handle.name - attribute.name [(index-specification)]
```

The *handle-name* is the handle of the *dialog-element-type* as defined in the HANDLE definition of the DEFINE DATA statement.

The *attribute-name* is the name of an attribute which has to be valid for the *dialog-element-type* of the handle.

Examples:

```
1 #PB-1 HANDLE OF PUSHBUTTON   /* #PB-1 is a handle-name of the
                               /* dialog-element-type PUSHBUTTON
RESET #PB-1.STRING...          /* #PB-1.STRING is the handle attribute operand
                               /* where STRING is a valid attribute-name of the
                               /* dialog-element-type PUSHBUTTON

1 #RB-1(1:5) HANDLE OF RADIOBUTTON /* #RB-1 is an array of five RADIOBUTTONs
IF #RB-1.CHECKED(3) = CHECKED      /* If the third radio-button control is
   THEN...                         /* checked ...
```

Querying, Setting and Modifying Attribute Values

In most applications, it will be necessary

- to set an attribute value before creating the dialog element,
- to modify the value after creating the dialog element, and
- to query an attribute value.

In some cases, it may be necessary to modify and query some attributes during processing, for example to query the checked/not checked state of a radio-button control or to disable (= modify) a menu item.

You can do that, for example, in the ASSIGN, MOVE or CALLNAT statements.

Examples:

```
1 #PB-1 HANDLE OF PUSHBUTTON      /* #PB-1 is a handle-name of the
...                               /* dialog-element-type PUSHBUTTON
#PB-1.STRING:= 'MY BUTTON'        /* Set or modify the value of the STRING
                                  /* attribute to 'MY BUTTON'
#TEXT:= #PB-1.STRING              /* Query the value of the STRING attribute
                                  /* and assign the value to #TEXT
CALLNAT 'SUBPGM1' #PB-1.STRING    /* Query the value of the STRING attribute
                                  /* and pass it on to the subprogram
```

When you use the *handle-name* variable only on the left side of the statements, as in the first of the three examples above, the attribute value is set or modified, that is, it is assigned the value of the specified *operand*.

When you use the *handle-name* variable on the right side of the statements, as in the second example, the attribute value is queried, that is, the value is assigned to the *operand*.

Once a handle has been defined (either explicitly in specified Natural code, or implicitly with the dialog editor), it can be used with most Natural statements. However, only a specific set of attributes can be queried, set or modified for a particular dialog element. To find out which values an attribute can have, see the chapter Attributes in the Dialog Components Manual.

Although an exact data type is specified for the values of most attributes, it is sufficient to supply move-compatible values to a handle attribute operand. The rules are the same as those for Natural variables.

Restrictions

Handle attribute operands must not be used in the following statements:

AT BREAK, FIND, HISTOGRAM, INPUT, READ, READ WORK FILE.

User-defined variables can be used instead.

Numeric/Alphanumeric Assignment

If you assign numeric operands to alphanumeric attributes, the values of these attributes will be in a non-displayable format. The Natural arithmetic assignment rules apply.

If you need a displayable format, you can use MOVE EDITED.

Examples:

```
#PB-1.STRING:= -12.34                         /* Non-displayable format
MOVE EDITED #I4 (EM = -Z(9)9) TO #PB-1.STRING /* Displayable format
```

The following edit masks may be used for the various format/length definitions of numeric operands:

Format/Length	**Edit Mask**
I1	-ZZ9
I2	-Z(5)9
I4	-Z(9)9
N*n.m*/P*n.m*	-Z(*n*).9(*m*)

How To Create and Delete Dialog Elements Dynamically

Dialog elements are usually added to a dialog by means of the dialog editor. However, they can also be created and deleted dynamically. This may be done, for example, when the layout of a dialog is strongly context-sensitive.

A dialog element is created dynamically with the ADD action of the PROCESS GUI statement. This action returns a handle to the newly created dialog element. As soon as the dialog element is created, this handle points to a set of attributes specified for the dialog element just created.

Note: ActiveX controls are created in a slightly different way than the standard way described below. This is described in Working with ActiveX Controls.

For more information on the actions available, and on the parameters that can be passed, see the chapter Executing Standardized Procedures.

Global Attribute List

By modifying any handle attribute operand of the form "*handlename.attributename*" (for example, #PB-1.STRING), you change an attribute value of the specific dialog element. As long as the dialog element is not yet created and the handle variable has its initial value (NULL-HANDLE), the handle attribute operand "*handlename.attributename*" refers to the global attribute list.

The global attribute list is a collection of all attributes defined for any dialog element. Natural contains one such collection. Whenever a dialog element is created, it "inherits" its attributes from this global attribute list. It does not inherit them when you create the dialog element with the PROCESS GUI statement action ADD using the WITH PARAMETERS option.

Creating Dialog Elements Statically and Dynamically

To define a dialog element statically (in the dialog editor), with an individual set of attributes, you must first set the attributes in the global attribute list to the desired values and then create the dialog element. After creation, the values of the attributes in the global attribute list remain intact. The next created dialog element gets the same attributes from the global attribute list as the previous one, except those that have been modified.

The status of the global attribute list as found in the "after open" event handler is influenced by the dialog elements defined statically. Therefore, before you start creating dialog elements dynamically in the "after open" event handler, you should reset the attributes by means of the PROCESS GUI action RESET-ATTRIBUTES to prevent your dialog elements from inheriting unexpected values from the global attribute list. If you want to avoid this inheritance problem, use the PROCESS GUI statement action ADD with the WITH PARAMETERS option.

Unexpected values may also result from having attribute values that mean different things if used by different types of dialog elements. For example, the value "s" of the attribute STYLE means "scaled" for the dialog element type bitmap control but "solid" for the dialog element type line control.

The PROCESS GUI action ADD is used to define a dialog element dynamically. This clause of the PROCESS GUI statement enables you to specify the attribute values within the statement. The inheritance of attributes from the global attribute list does not affect the PROCESS GUI statement action ADD. The attributes specified in the statement are transferred to the global attribute list before the action ADD is performed.

Note: When you use the PROCESS GUI statement with Parameter Clause 2 of the ADD action, the global attribute list is not used or affected. For parameters which are needed to create the dialog element, but which were not specified in the WITH PARAMETERS section of the PROCESS GUI action ADD statement, the default value is taken. Apart from these, only the parameters which are passed explicitly in the parameter list are used to create the dialog element.

To create list-box and selection-box items dynamically, it may be more convenient to use the PROCESS GUI action ADD-ITEMS. This allows you to insert several items at a time.

Example:

```
/* #PB-A inherits the current settings of the global attribute list
#PB-A.STRING := ’TEST1’
PROCESS GUI ACTION ADD WITH #DLG$WINDOW PUSHBUTTON #PB-A
#PB-B.STRING := ’TEST2’
/* #PB-B has the same attributes as #PB-A except STRING. This leads to #PB-B
/* covering #PB-A.
PROCESS GUI ACTION ADD WITH #DLG$WINDOW PUSHBUTTON #PB-B
COMPUTE #PB-C.RECTANGLE-Y = #PB-B.RECTANGLE-Y + #PB-C.RECTANGLE-H + 20
/* #PB-B has the same attributes as #PB-A except RECTANGLE-Y
/* #PB-C will be located 20 pixels below #PB-B
PROCESS GUI ACTION ADD WITH #DLG$WINDOW PUSHBUTTON #PB-C
```

To delete dialog elements dynamically, you use the PROCESS GUI action DELETE. You can also use this technique to delete dialog elements created with the dialog editor (at design time). You should, however, avoid using the handle of the deleted dialog element because this is invalid.

Dialog elements often do not have to be created dynamically. In some cases, it is sufficient to make dialog elements VISIBLE = TRUE and VISIBLE = FALSE, depending on the context. This technique is more efficient and easier to handle. It also enables you to "insert" dialog elements anywhere in the navigation sequence.

Example:

```
DEFINE DATA LOCAL
   ...
   1 #PB-1 HANDLE OF PUSHBUTTON
   ...
END-DEFINE
...
#PB-1.VISIBLE := FALSE
...
IF...                        /* Logical condition
   #PB-1.VISIBLE := TRUE
END-IF
```

How to Handle Events of Dynamically Created Dialog Elements

When a dialog element is created dynamically, you cannot use the dialog editor to associate events to it. Instead, you must handle all events of all dynamically created dialog elements in the DEFAULT event. In this event, you must filter out which event occurred for which dialog element. The code for this is similar to the code generated by the dialog editor. The general structure is:

Example:

```
DECIDE ON FIRST *CONTROL
VALUE #PB-A
   DECIDE ON FIRST *EVENT
      VALUE 'CLICK'
         /* Click event-handler code
      NONE
      IGNORE
   END-DECIDE
VALUE #PB-B
   ...
VALUE #PB-C
   ...
END-DECIDE
```

In the case of event code for dynamically created ActiveX controls, *where event parameters are used*, it is necessary to precede the event code with an OPTIONS 2 statement containing the name of the event, otherwise the compiler will not be able to process parameter references (e.g., #OCX-1.<<PARAMETER->>) successfully. However, in contrast to the implicit generation of the OPTIONS statement by the Dialog Editor for events for statically created controls, no OPTIONS 3 statement should be coded in this case. Otherwise the Dialog Editor would falsely interpret the OPTIONS 3 statement as the end marker for the DEFAULT event, resulting in a scanning error on attempting to load the dialog.

Example:

```
DECIDE ON FIRST *CONTROL
VALUE #OCX-1  /* MS Calendar control
  DECIDE ON FIRST *EVENT
   VALUE '-602' /* DispID for KeyDown event
     OPTIONS 2 KeyDown
      /* KeyDown event-handler code containing parameter
      /* access (e.g. #OCX-1.<>)
     NONE
      IGNORE
 END-DECIDE
...
END-DECIDE
```

How To Enable and Disable Dialog Elements

During end-user interaction, it may be implicitly clear that certain dialog elements must not be used. For example, if a dialog requiring personnel data contains a group of radio-button controls for marital status and an input-field control for date of marriage, the input-field control must be disabled whenever the marital status is other than "married".

There are two ways to do this:

- Use Natural code to enable/disable a dialog element dynamically.
- Use the dialog editor (to disable a dialog element initially).

The first method is used more often.

The Natural code might look like this:

Examples:

```
/*First alternative
...
IF #RB-1.ENABLED = TRUE     /* Logical condition
   #IF-1.ENABLED := TRUE    /* Set ENABLED to TRUE
END-IF
...
/*Second alternative
#PB-1.ENABLED := #RB-1.ENABLED
```

When you use the dialog editor, you set the attribute ENABLED to TRUE by marking the "Enabled" entry in the dialog element’s attributes window.

To disable editing in input-field controls, selectionbox controls and edit area controls, it is not always necessary to disable these dialog elements entirely. It may be sufficient to make them MODIFIABLE = FALSE.

Defining and Using Context Menus

As from Natural v4.1.1, it is possible to create context menus for use within Natural applications. The context menus can be completely static (i.e., the menu contents are known in advance and can be built via the dialog editor) or wholly or partially dynamic (i.e., the menu contents and/or state depend on the runtime context and are not completely known at design time).

Construction

A context menu is very similar in concept to a submenu. Therefore, the same menu editor is used for editing a context menu as is used for editing a dialog’s menu bar. Menu items can be added to context menus, and events associated with them, in exactly the same way as for menu-bar submenus. There are no functional differences to the menu-bar editor, except that the ’OLE’ combo box (which is applicable only to top-level menu-bar submenus) will always be disabled. It should be noted, however, that any accelerators defined for context menu items will be globally available as long as that menu item exists. Furthermore, the accelerator will trigger the menu item for which it is defined even if the context menu is not being displayed or if the focus is on a control using a different context menu or no context menu at all.

The context-menu editor may be invoked via either a new menu item, ’Context menus...’ on the ’Dialog’ menu, or via its associated accelerator (CTRL+ALT+X by default), or toolbar icon. However, because the context-menu editor can only edit one context menu editor at a time, the context-menu editor is not invoked directly. Instead, the Dialog Context Menus window is shown, where operations on the context menu as a whole are made, and from which the menu editor for a given (selected) context menu can be invoked.

Internally, in order to distinguish between submenus and context menus, context menus have a new type, CONTEXTMENU. Otherwise, the generated code in both cases is identical. Here is some sample code illustrating the statements used to build up a simple context menu containing two menu items:

```
/* CREATE CONTEXT MENU ITSELF:
PROCESS GUI ACTION ADD WITH PARAMETERS
  HANDLE-VARIABLE = #CONTEXT-MENU-1
  TYPE = CONTEXTMENU
  PARENT = #DLG$WINDOW
END-PARAMETERS GIVING *ERROR
/* ADD FIRST MENU ITEM:
PROCESS GUI ACTION ADD WITH PARAMETERS
  HANDLE-VARIABLE = #MITEM-1
  TYPE = MENUITEM
  DIL-TEXT = 'Invokes the first item'
  PARENT = #CONTEXT-MENU-1
  STRING = 'Item 1'
END-PARAMETERS GIVING *ERROR
/* ADD SECOND MENU ITEM:
PROCESS GUI ACTION ADD WITH PARAMETERS
  HANDLE-VARIABLE = #MITEM-2
  TYPE = MENUITEM
  DIL-TEXT = 'Invokes the second item'
  PARENT = #CONTEXT-MENU-1
  STRING = 'Item 2'
END-PARAMETERS GIVING *ERROR
```

Note that if context menus or context-menu items are created dynamically in user-written code, the context menu or menu items will not be visible to the dialog editor. For example, the dynamically created menu item will not be visible in the context-menu list box, and the dynamically created menu items will not be visible in the context-menu editor.

Association

After creating a context menu, the context menu needs to be associated with a Natural object. Context menus are supported for almost all controls types capable of receiving the keyboard focus and for the dialog window itself. The full list includes ActiveX controls, bitmaps, canvasses, edit areas and input fields, list boxes, push buttons, radio buttons, scroll bars, selection boxes, table controls, toggle buttons, standard and MDI child windows, and MDI frame windows.

For all object types supporting context menus, the corresponding attribute dialogs in the dialog editor include a read-only combo box listing all context menus created by the dialog editor, together with an empty entry. The selection of the empty entry implies that no context menu is to be used for this object, and is the default.

Internally, the association is achieved by a new attribute, CONTEXT-MENU , which should be set to the handle of a context menu. This attribute can be assigned at or after object creation time, and defaults to NULL-HANDLE if not specified, indicating the absence of a context menu. For context menus created by the dialog editor, the context menu is specified at control creation time as illustrated below:

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
  HANDLE-VARIABLE = #LB-1
  TYPE = LISTBOX
  RECTANGLE-X = 585
  RECTANGLE-Y = 293
  RECTANGLE-W = 142
  RECTANGLE-H = 209
  MULTI-SELECTION = TRUE
  SORTED = FALSE
  PARENT = #DLG$WINDOW
  CONTEXT-MENU = #CONTEXT-MENU-1
  SUPPRESS-FILL-EVENT = SUPPRESSED
END-PARAMETERS GIVING *ERROR
```

The same syntax can also be used for controls created in user-written event code. In other cases, where the control was created by the dialog editor but the context menu was not, the context menu attribute must be assigned to the control after its creation, e.g., in the dialog’s AFTER-OPEN event:

```
/* CONTEXT MENU SPECIFIED AFTER CREATION:

#LB-2.CONTEXT-MENU := #CONTEXT-MENU-2
```

Note that a context menu is not destroyed when an object using it is destroyed. Instead, it is destroyed when its parent object (typically, the dialog for which the context menu was defined) is destroyed. Similarly, the assignment of a new menu handle to the CONTEXT-MENU attribute where one is already assigned does not result in the previous context menu being destroyed. Thus, using the above examples, neither of the following statements results in CONTEXT-MENU-1 being destroyed:

```
PROCESS GUI ACTION DELETE WITH #LB-1            /* #CONTEXT-MENU-1 LIVES ON

#LB-1.CONTEXT-MENU := #CONTEXT-MENU-2           /* SAME HERE
```

Invocation

The invocation of static context menus is transparent to the application. The tracking of the context menu and the triggering of the events associated with the menu items is done by Windows and Natural. The context menu is always displayed at the current mouse cursor position. Therefore, there are no new PROCESS GUI statements for displaying context menus.

However, in order to support dynamic context menus or static context menus that need to be modified at runtime (e.g. to disable or check particular menu items before the context menu is displayed), context menus and submenus receive a BEFORE-OPEN event. This applies to submenus belonging to a menu bar as well as to submenus belonging to a context menu. In addition, it is possible to suppress this event via the use of a new attribute, SUPPRESS-BEFORE-OPEN-EVENT, which defaults to SUPPRESSED. Assuming the event is not suppressed, the BEFORE-OPEN event will be triggered immediately before a context menu is displayed. This gives the application the chance to modify the context menu according to the current program state. For example, menu items could be added or deleted, or particular menu items grayed or checked. Here is some sample code for the BEFORE-OPEN event:

```
/* DELETE FIRST MENU ITEM:
PROCESS GUI ACTION DELETE WITH #MITEM-1
/* CHECK SECOND MENU ITEM:
#MITEM-2.CHECKED := CHECKED
/* DISABLE THIRD MENU ITEM:
#MITEM-3.ENABLED := FALSE
/* INSERT NEW MENU ITEM BEFORE #MITEM-3:
PROCESS GUI ACTION ADD WITH PARAMETERS
  HANDLE-VARIABLE = #MITEM-4
  TYPE = MENUITEM
  DIL-TEXT = 'Invokes the first item'
  PARENT = #CONTEXT-MENU-1
  STRING = 'Item 3'
  SUCCESSOR = #MITEM-3
END-PARAMETERS GIVING *ERROR
```

For context menus not created by the dialog editor, the handling of the BEFORE-OPEN event must be done in the DEFAULT event for the dialog. Note also that if a control or dialog is disabled, no context menu is displayed, and the BEFORE-OPEN event is also not triggered. The same applies if the context menu itself is disabled. For example:

```
#CONTEXT-MENU-1.ENABLED := FALSE        /* DISABLE CONTEXT MENU DISPLAY
```

Note that it is possible to disable the context menu in this way during the BEFORE-OPEN event, allowing selective disabling of the context menu depending on the mouse cursor position within the control. For example, it might be desired to only display a context menu if the mouse cursor is over a selected list-box item. Determining whether this is the case is possible via the use of two PROCESS GUI ACTION calls:

- INQ-CLICKPOSITION has been extended to controls other than bitmaps and canvasses to return the (X, Y) position of the right mouse button click within the control. This is updated immediately prior to the sending of the BEFORE-OPEN event.
- INQ-ITEM-BY-POSITION. This allows translation of the relative co-ordinate returned by INQ-CLICKPOSITION applied to a list box to the corresponding item.

As an example of the use of these two new actions, consider the situation where we want to detect whether the cursor was over a selected list-box item when the right mouse button was pressed in order to determine whether to display a context menu or not. This can be achieved by the following code in the BEFORE-OPEN event of the associated context menu:

```
PROCESS GUI ACTION INQ-CLICKPOSITION WITH
   #LB-1 #X-OFFSET #Y-OFFSET
PROCESS GUI ACTION INQ-ITEM-BY-POSITION WITH
   #LB-1 #X-OFFSET #Y-OFFSET #LBITEM
#MENU = *CONTROL
IF #LBITEM = NULL-HANDLE                 /* NO ITEM UNDER (MOUSE) CURSOR */
  #MENU.ENABLED := FALSE
ELSE
  IF #LBITEM.SELECTED = FALSE            /* ITEM UNDER CURSOR DESELECTED */
     #MENU.ENABLED := FALSE
  ELSE                                   /* ITEM UNDER CURSOR IS SELECTED */
     #MENU.ENABLED := TRUE
  END-IF
END-IF
```

In some cases, it may be desired to automatically select the item under the mouse cursor if it is not already selected, clearing any existing selection. For list boxes, it is possible to achieve this by using the new AUTOSELECT attribute, either directly or via the new 'Autoselect' check box (see previous bitmap) in the List Box Attributes window in the dialog editor. If this attribute is set to TRUE, Natural will automatically update the selection before sending the BEFORE-OPEN event, if the context menu was invoked over an unselected list-box item.

For table controls, any change in the selection must be done via the application itself in the BEFORE-OPEN event. To make this possible, another new PROCESS GUI ACTION has been introduced for table controls:

- TABLE-INQUIRE-CELL. This returns the cell's row and column number (starting from 1) for a relative (X, Y) position within the table. This position can (and would typically be) the position returned by a previous call to PROCESS GUI ACTION INQ-CLICKPOSITION.

Sharing of Context Menus

It is of course possible to associate the same context menu with more than one object (i.e., control or dialog). For example:

```
#LB-1.CONTEXT-MENU := #CTXMENU-1
#LB-2.CONTEXT-MENU := #CTXMENU-1
```

In such a scenario, we need to be able to determine for which control the context menu was invoked. We cannot use *CONTROL in the BEFORE-OPEN event, because this will contain the handle of the context menu itself. Instead, it is necessary to inquire which control has the focus, since Natural automatically places the focus on the control for which the context menu is being invoked. Here is some sample BEFORE-OPEN event code illustrating the use of this technique:

```
PROCESS GUI ACTION GET-FOCUS WITH #CONTROL
DECIDE ON FIRST VALUE OF #CONTROL
  VALUE #LB-1
     #MITEM-17.ENABLED := FALSE
  VALUE #LB-2
     #MITEM-17.CHECKED := CHECKED
  NONE
     IGNORE
END-DECIDE
```

System Variables

Whenever you specify an event to occur with a given dialog element, the dialog editor generates code containing the Natural system variables *CONTROL, *DIALOG-ID and *EVENT.

During the processing, *CONTROL contains the dialog element’s handle, *EVENT contains the event name and *DIALOG-ID identifies an instance of a dialog.

You can reference these system variables whenever you enter Natural code within the dialog editor. If, for example, the end user clicks on a push-button control and the event handler calls a shared subroutine, you can use these system variables as logical condition criteria to trigger the subroutine.

For further details on these system variables, see the Natural Reference Manual.

Generated Variables

#DLG$PARENT

You use this generated variable of type "user" to work with MDI child windows, for example. When you create a dialog, Natural generates this variable in order to hold the handle of the parent dialog. In event-handler code, you can, for example, use this variable to open an MDI child dialog from another MDI child dialog, as shown below.

Note: You should not use names for user-defined variables that begin with #DLG$ to avoid conflicts with generated variables.

Example:

```
OPEN DIALOG 'MDICHILD' #DLG$PARENT #CHILD-ID
```

#DLG$WINDOW

You use this generated variable to dynamically set the attributes within a dialog. When you create a dialog, Natural generates this variable in order to hold the handle of the dialog window. #DLG$WINDOW is the default name of this variable; you may change it by overwriting the "Name" entry in the upper left of the dialog's attributes window. In event-handler code, you can, for example, use this variable to minimize the dialog window if certain logical condition criteria are met, as shown below.

#DLG$WINDOW represents the graphical user interface aspects of a dialog, while the *DIALOG-ID system variable represents the runtime aspects. *DIALOG-ID must be used in OPEN DIALOG, CLOSE DIALOG and SEND EVENT statements.

Note: You should not use names for user-defined variables that begin with #DLG$ to avoid conflicts with generated variables.

Example:

```
...
IF ...
   #DLG$WINDOW.MINIMIZED := TRUE
END-IF
...
```

Message Files and Variables as Sources of Attribute Values

Most dialog elements have a STRING attribute. As an alternative to specifying the attribute value by typing in the text in the "String" entry of the attributes window, you can select a variable or a message file number from which the text is taken at runtime. In this case, the attribute value is determined by the variable’s current value or the selected message file at the dialog element’s creation time. You can also specify attribute sources for the BITMAP-FILE-NAME, DIL-TEXT and ACCELERATOR attributes.

To select a message file number or specify a variable

1. Invoke the dialog element’s attribute window.
2. Push the "Source" button to the right of the "String" entry.

The "Attribute Source" dialog box appears. The default attribute source is "Constant"; you can also enter the number of the message file, or enter the variable name.

Note: If you are using an integer variable as the source of an attribute value, note that at runtime, the message with the corresponding number from your message file will be displayed. To avoid this, you can MOVE the contents of this integer variable to a variable of format N, for example.

Triggering User-Defined Events

Aside from standard events, such as before-open, you may define user-defined events for dialogs. User-defined events are useful whenever it is necessary for one dialog to cause an action to occur in another dialog.

A user-defined event occurs whenever you have specified a SEND EVENT statement in dialog A with the name of a user-defined event in the target dialog B. This target dialog B for which you wish to trigger the user-defined event must already be active. You can activate dialog B by using the OPEN DIALOG statement. If you do not issue the OPEN DIALOG statement first, the SEND EVENT statement will cause a runtime error.

You can define your own events for dialogs by pressing the "New" button in the "Events" dialog event handler menu or from the dialog’s context menu. Enter any name for your newly-defined event and specify the corresponding event section. It is recommended that this name begin with "#" to distinguish your event from predefined events.

During execution of an event handler, the SEND EVENT statement triggers the user-defined event handler in a different dialog. After this user-defined event handler has been executed, control will be returned to the previous dialog, whose execution will resume at the statement following the SEND EVENT statement. This can be compared to a CALLNAT statement that causes a subprogram to be executed.

Similar to the OPEN DIALOG statement, parameters may be passed to the dialog. In order to pass parameters selectively (*PARAMETERS-clause*), you have to specify the name of the dialog in addition to the identifier of the dialog *(operand2)*.

The SEND EVENT statement must not trigger an event in a dialog that is about to process an event. This is the case, for example, when dialog A sends an event to dialog B and the event handler in dialog B sends an event to dialog A which has not yet finished its event handling. A similar case is when dialog A opens dialog B and the before-open or after-open event contains a SEND EVENT back to dialog A.

To trigger a user-defined event, you specify the following syntax:

```
SEND EVENT operand1 TO [DIALOG-ID] operand2
          [{ WITH operand3...                                     }]
          [{ USING [DIALOG] 'dialog-name' WITH PARAMETERS-clause }]
```

Operands

Operand1 is the name of the event to be sent.

Operand2 is the identifier of the dialog receiving the user-defined event and must be defined with format/length I4. You can retrieve this identifier, for example, by querying the value of #DLG$PARENT.CLIENT-DATA.

Passing Parameters to the Dialog

It is possible to pass parameters to the dialog receiving the user event.

As *operand3* you specify the parameters which are passed to the dialog.

With the *PARAMETERS-clause*, parameters may be passed selectively.

PARAMETERS-clause

```
PARAMETERS [parameter-name =operand3 ]... END-PARAMETERS
```

Note: You may only use the PARAMETERS-clause if the target dialog is cataloged.

Dialog-name is the name of the dialog receiving the user-defined event.

When you use only *operand3* to pass parameters, it might look like this:

Example:

```
/* The following parameters are defined in the dialog’s
/* parameter data area:
1 #DLG-PARM1 (A10)
1 #DLG-PARM2 (A10)
1 #DLG-PARM3 (A10)
1 #DLG-PARM4 (A10)
/* When sending the user-defined event, pass the operands #MYPARM1 ’MYPARM2’ to
the parameters #DLG-PARM1 and #DLG-PARM2:
SEND EVENT ’MYEVENT’ TO #DLG$DIA-ID WITH #MYPARM1 ’MYPARM2’
```

When you use the *PARAMETERS-clause*, the user-defined event might look like this:

Example:

```
/* The following parameters are defined in the dialog’s
/* parameter data area:
1 #DLG-PARM1 (A10)
1 #DLG-PARM2 (A10)
1 #DLG-PARM3 (A10)
1 #DLG-PARM4 (A10)
/* When sending the user-defined event, the operand #MYPARM2 is passed to the
/* parameter #DLG-PARM2 and the operand ’MYPARM3’ is passed to the parameter
/* #DLG-PARM3:
SEND EVENT ’MYEVENT’ TO #DLG$DIA-ID
  USING DIALOG ’MYDIALOG’
  WITH PARAMETERS
    #DLG-PARM3=’MYPARM3’
    #DLG-PARM2=#MYPARM2
  END-PARAMETERS
```

To avoid format/length conflicts between operands passed and their parameter definitions, see the BY VALUE option of the DEFINE DATA statement in the Natural Statements Manual.

Suppressing Events

If an event occurs, normally an event handler will be triggered. It may, however, sometimes be necessary to dynamically suppress the execution of the event-handler code whenever the event has occurred. For example, if you want to modify the string of an input field control within the change-event handler, you must suppress the change event before modification to avoid an infinite loop because the modification itself triggers a change event.

The event-handler code may look like this:

Example:

```
...
IF...                                            /* Logical condition criteria
   #IF-1.SUPPRESS-CHANGE-EVENT := SUPPRESSED     /* Suppress the event
END-IF
...
```

By default, the dialog editor generates code to suppress all events for which no event handler code has been entered. In the dialog editor, you can also suppress an event with the Suppress option in the "Events..." dialog box.

If you suppress an event, the before-any and after-any events are also suppressed for this event.

Menu Structures, Toolbars and the MDI

Creating a Menu Structure

A menu structure consists of three types of dialog elements:

- menu-bar controls,
- menu items,
- submenu controls.

A menu structure has one menu-bar control consisting of several menu items. The menu bar with its items is displayed directly beneath the window's title bar. Each menu item may be simple or may represent a submenu control, which allows you to pull down several menu items grouped vertically. Therefore, submenu controls may contain items representing a submenu control one level lower. A submenu control becomes visible when the representing item in the menu-bar control or the parent submenu control is clicked upon.

There are two ways to create menu structures:

- Use the dialog editor; or
- use Natural code.

If you use the dialog editor

1. Check the "Menu Bar" entry in the dialog's attribute window. Click OK.
   When you go back to the dialog, a dummy menu-bar control appears.
2. Double-click on the dummy menu-bar control, or from the Natural Menu, select "Dialog > Menu Bar", or use CTRL+M.
   The "Dialog Menu Bar" dialog box appears. This dialog box is divided into three group frames: menu bar, selected submenu and selected menu item.
3. In the selected menu items group frame, use "New" to add a menu item behind the selected position, or at the beginning. Now use the selected menu-item group frame to modify attribute values or add event handlers to the new menu item.

Normal menu items have a click event whose code is executed when the end user clicks on the menu item.

Note: The MENU-ITEM-TYPE of the menu item can also be "Separator", in which case the item is no text item.

If you use Natural code

1. Create a Menu Bar with the PARENT attribute set to 'NULL-HANDLE' or '*windowhandle*'.
2. To create a simple menu item: the PARENT attribute must have the value '*menubarhandlename*'.
3. To create a submenu control: the submenu control's PARENT attribute must have the value 'NULL-HANDLE' or '*windowhandlename*'. Then create a menu item with PARENT = '*menubarhandlename*' and MENU-HANDLE = *'submenuhandlename'*.
4. Then associate the menu bar with a dialog window by updating the window's MENU-HANDLE attribute with the handle of the menu bar as set in the first step.
5. The event handling for the dynamically created menu items must be done in the default event handler, as described in the section How to Create and Delete Dialog Elements Dynamically.

The PARENT attribute determines when the menu bar or the submenu control will be destroyed. When PARENT = '*windowhandlename*', the menu bar/the submenu control will be destroyed when the window is destroyed. This is the default setting, which is also used by the dialog editor. If PARENT = NULL-HANDLE, the menu bar/the submenu control will be destroyed only when the application is terminated.

If you define the menu structure’s handles inside a global data area, you can share these definitions among several dialogs.

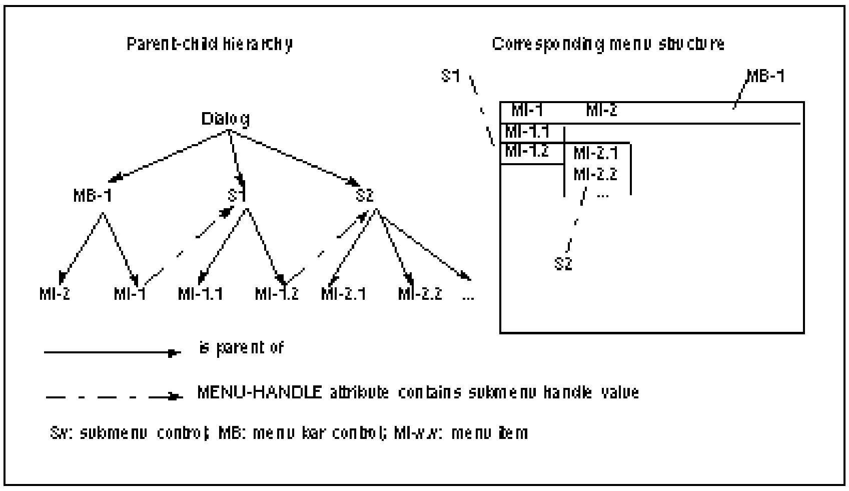
To build the above menu structure

1. Define the handles of the menu-bar control, the menu items, and the submenu control(s) as the user-defined variables in the handler of the applicable event.
2. Create the controls and items by assigning values to the attributes (PARENT, ...) and by executing the PROCESS GUI statement action ADD.
3. Create the controls and items in the sequence menu-bar control, submenu control with menu items.
4. Insert the controls and items in the sequence submenu control into menu-bar control, and menu-bar control into dialog window.

You can study how to build a menu structure in code by using the enhanced dialog list mode to list a dialog with an editor-built menu. To get a code model for creating a menu item, create a menu-bar control with the dialog editor, go to the menu-bar control attributes window, cut a menu item and paste it into any chosen event-handler section. The generated code for the menu item appears.

Parent-Child Hierarchy in Menu Structures

Sometimes, it is necessary to use code for going through each element in a menu structure. For menus, the parent-child hierarchy is structured in a way that is not evident from the graphical representation of the menu structure.


Parent-child hierarchy
Corresponding menu structure
Dialog
MB-1
S1
S2
MI-2
MI-1
MI-1.1
MI-1.2
MI-2.1
MI-2.2
...
is parent of
MENU-HANDLE attribute contains submenu handle value
Sx: submenu control; MB: menu bar control; MI-x.x: menu item

In the above diagram, the first child of the dialog would be the menu-bar control. Its successor would be submenu control S1, and so on. To go from menu item MI-1 to submenu S1, you query for the MENU-HANDLE attribute value of MI-1. The value you get is the handle value of S1.

Creating a Toolbar

There are two ways of creating toolbars and their items:

- Use the dialog editor; or
- use Natural code to create them dynamically.

To use the dialog editor

1. Double-click on the toolbar or from the Natural Menu, select "Dialog > Toolbar".
   The toolbar attributes window opens.
2. Add toolbar items by clicking on the "New" push button.
3. Assign bitmap file names and other attribute values to the new toolbar item.

If you want to use Natural code for dynamic creation, you can study how to build a tool bar in code. Use the enhanced dialog list mode to list a dialog with an editor-built tool bar.

Sharing Menu Structures, Toolbars and DILs (MDI Application)

An MDI (multiple document interface) application consists of a frame dialog that provides the menu structure, toolbar, and DIL shared among all child dialogs. An MDI frame dialog allows you to tile or cascade its child dialogs.

Note: You may only share the toolbar if the PARENT of the toolbar is the dialog of the highest level (the main dialog of an application).

To create an MDI frame dialog

1. Use the dialog editor, and go to the dialog object’s attributes window.
2. Choose "MDI frame window" in the "Type" entry.

An MDI frame dialog must not contain dialog elements other than menu-bar control, submenu control, menu item, toolbar, and toolbar item.

To create an MDI child dialog

1. Use the dialog editor, and go to the dialog object’s attributes window.
2. Choose "MDI child window" in the "Type" entry.

An MDI child dialog:

- can be moved and sized only inside the area of their MDI frame dialog;
- can be maximized to the full size of the area of their MDI frame dialog;
- can be minimized, after which its icon appears at the bottom of its MDI frame dialog;
- can have its own menu structure, toolbar, and DIL. Those do not appear inside the child dialog but are displayed in the MDI frame dialog when the child dialog is active. When another MDI child dialog becomes active, the menu structure, toolbar, and DIL change at the same time;
- can be arranged in a tile or cascade by setting a menu item’s attribute MENU-ITEM-TYPE to the values "MDI Cascade" or "MDI Tile";
- can have its title added to the end of an MDI-WINDOWMENU type submenu control. By choosing one of these menu items, the corresponding MDI child dialog becomes active.

If you want to open an MDI child dialog from within an MDI frame dialog, you can, for example, create a menu item in a menu structure of an MDI frame dialog and define a click event for the menu item. You then write the OPEN DIALOG code for opening an MDI child dialog in the click event handler. The end user will open the MDI child dialog from within the MDI frame dialog by clicking on the menu item, triggering the click event handler.

Example:

```
OPEN DIALOG ’MDICHILD’ #DLG$WINDOW #CHILD-ID
```

The first operand is the name of the dialog created by the dialog editor by selecting "MDI child window" in the "Type" selection box. The second operand is the parent of the new MDI child dialog. This must be the MDI frame dialog. The third operand is a Natural variable defined as I4 in the dialog’s data areas. This variable receives the dialog ID returned by the system.

Note: #DLG$WINDOW is a generated variable.

You can also open an MDI child dialog from within another MDI child dialog (open a sibling of your MDI child dialog). Then you write a similar click-event handler as above:

Example:

```
 OPEN DIALOG ’MDICHILD’ #DLG$PARENT #CHILD-ID
```

The first and the third operands are the same as above. The second operand must be the parent of both MDI child dialogs.

Note: #DLG$PARENT is a generated variable.

Executing Standardized Procedures

For procedures frequently needed in event-driven applications, the following is available:

- a set of PROCESS GUI statement actions and
- a set of NGU-prefixed subprograms and dialogs in library SYSTEM.

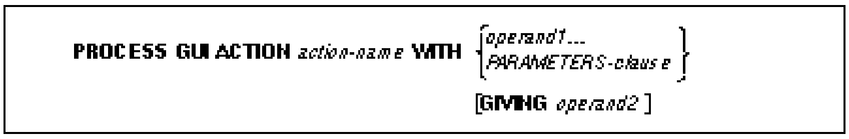
Examples for frequently needed procedures are starting up a message box, reading the lines entered into an edit area control, or dynamically creating dialog elements.

For your convenience, the local data areas NGULKEY1 and NGULFCT1 are automatically included in the list of local data areas used by any new dialog.

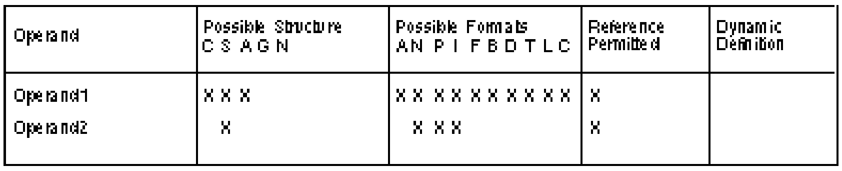
- NGULFCT1 is necessary to use the NGU-prefixed subprograms and dialogs;
- NGULKEY1 lists reserved keywords to be used in any event-handler code. This enables you to refer to certain attribute values by the more meaningful keyword rather than by the numeric IDs. It also enables you to use meaningful dialog element names as parameters.

For more information on the PROCESS GUI statement actions, subprograms and dialogs available, and on the parameters that can be passed, see the chapter Executing Standardized Procedures of the Natural Dialog Components Manual.

PROCESS GUI Statement



```
                                          ┌ operand1...        ┐
PROCESS GUI ACTION action-name WITH      ┤                     ├
                                          └ PARAMETERS-clause  ┘
                                          [GIVING operand2 ]
```



Operand	Possible Structure					Possible Formats										Reference Permitted	Dynamic Definition
	C	S	A	G	N	A	N	P	I	F	B	D	T	L	C		
Operand1	X	X	X			X	X	X	X	X	X	X	X	X	X	X	
Operand2		X					X	X	X							X	

The PROCESS GUI statement is used to perform an action. An action in this context is a procedure frequently needed in event-driven applications.

As *action-name*, you specify the name of the action to be invoked.

As *operand1*, you specify the parameter(s) to be passed to the action. The parameters are passed in the sequence in which they are specified.

For the action "ADD", you can also pass parameters by name (instead of position); to do so, you use the *PARAMETERS-clause*:

```
PARAMETERS |parameter-name =operand1 |_ END-PARAMETERS
```

This clause can only be used for the action "ADD", not for any other action.

As *operand2*, you can specify a field to receive the response code from the invoked action after the action has been performed.

Linking Dialog Elements to Natural Variables

In cases where you want to map database fields or other program variables to the user interface, input-field controls and selection-box controls may be linked to Natural variables. This makes it easier to modify and query them.

If the end user has entered data in an input-field control or a sebox control and sets the focus to another dialog element, a leave event occurs and the content (STRING) is moved to the variable. Thus, the variable is updated. Note that the variable will *not* be updated if the end user enters data and a change event occurs.

To refresh the content of the dialog element after the linked variable has been modified in code

Use the PROCESS GUI statement action REFRESH-LINKS.

Modifying and querying input-field controls with the ASSIGN statement would normally work like this:

Example:

```
...
#IF-1.STRING := ’12345’
#TEXT := #IF-1.STRING
...
```

However, you can also link a Natural variable to the input-field control or selection box control. You can also link an indexed variable to a dialog element or an array of dialog elements.

To link a variable in Natural code, set the attribute LINKED to TRUE and modify the attribute VARIABLE by setting it to the Natural variable name:

Example:

```
...
#IF-1.LINKED := TRUE
#IF-1.VARIABLE := MYVARIABLE
...
```

To use the dialog editor to enter the name of the Natural variable

1. Double-click on your input-field control.
   The corresponding attributes window appears.
2. Click on the "Source" push button to the right of the "String" entry.
   The "Source for *handlename*" dialog box appears.
3. Choose "Linked variable".
4. Enter the variable name (such as MYVARIABLE in the example above).

There are two possibilities to link an indexed variable such as "MYVARIABLE (A20/1:5)":

- you link a single dialog element to the indexed variable; then you specify the index, such as "MYVARIABLE(2)" in the variable name field of the "Source for *handlename*" dialog box, or
- you link an array of dialog elements to the indexed variable; then you do not specify an index in the variable name field. In this case, the occurrences of the array and the index of the variable must be compatible. "MYVARIABLE (A20/1:5)" could be linked to a one-dimensional array with up to five occurrences.

Validating Input in a Dialog Element

If an input-field control or a sebox control is linked to a Natural variable, this dialog element may be checked automatically when it loses the focus to another dialog element in the same dialog. This enables you to validate the end user’s input. An input field control or a sebox control will not be checked when the end user clicks on a menu item or switches to another application.

If you specify an edit mask with one of these two dialog elements, the field content is checked against this edit mask plus the Natural data type of the linked variable.

If no edit mask is specified, the field content is checked against the Natural data type only.

There are two ways of specifying an edit mask in an input-field control or a selection box control:

- Use Natural code; or
- use the dialog editor.

The Natural code might look like this:

Examples:

```
...
/* Create an input-field control
   1 #IF-1 HANDLE OF INPUTFIELD
...
/* Assign the Edit Mask
#IF-1.EDIT-MASK := ’999’
```

To specify the edit mask with the dialog editor

Open the input-field control’s attribute window and use the "Edit Mask" entry.

When the field check fails, a message box comes up where the end user can choose "Retry" or "Cancel". "Retry" means that the entered text string remains unchanged and can be corrected. "Cancel" means that the field is reset to the current content of the linked variable.

Storing and Retrieving Client Data for a Dialog Element

For a number of dialog elements, the CLIENT-DATA attribute may hold an arbitrary I4 value. This may be useful for linking data to a specific dialog element. A list-box item, for example, can receive and pass on the ISN of a database record. The CLIENT-DATA attribute value may be changed at any time.

In Natural code, this might look like this:

Example:

```
DEFINE DATA
LOCAL
  1 #LBITEM-1 HANDLE OF LISTBOXITEM

  1 #ISN (I4)
   ...
END-DEFINE
...
READ...
   #LBITEM-1.CLIENT-DATA:= #ISN
END-READ
...
```

Note: The CLIENT-DATA attribute of a dialog is reserved for its dialog ID.

Client data may also be set and retrieved as alphanumeric string. In this case, you use the CLIENT-KEY and CLIENT-VALUE attributes in combination.

1. You first assign a value to the CLIENT-KEY attribute. This determines the key under which the string is to be stored for a dialog element.
2. You then assign an alphanumeric string to the CLIENT-VALUE attribute of the dialog element.

This enables you to store a number of key/value pairs for one dialog element.

Example:

```
#LB-1.CLIENT-KEY:= 'ANYKEY'
#LB-1.CLIENT-VALUE:= 'ANYSTRING'        /* The string to be stored
```

To query a dialog element for a particular string

1. You first assign a CLIENT-KEY value to the dialog element.
2. Then you query the dialog element for the corresponding CLIENT-VALUE.

If you assign a value to the CLIENT-KEY of a dialog element, this value is also valid for subsequent querying and modifying of other dialog elements.

If you query the CLIENT-VALUE of a CLIENT-KEY and there is no such pair among the key/value pairs of the dialog element, an empty string (' ') is returned.

It is advisable to reuse keys that are not needed because you may use only a limited number of keys.

Example:

```
#LB-1.CLIENT-KEY:= ’ANYKEY’
IF #LB-1.CLIENT-VALUE EQ ’ANYSTRING’ THEN
...
END-IF
```

Creating Dialog Elements on a Canvas Control

You can use a canvas control as a background to draw the following dialog elements on it: the rectangle, line and graphictext controls. These dialog elements "visualize" information. You can, for example, create three or four rectangle controls, fill them with color and change their size at runtime. This way, you can build your own bar chart.

Once you have created a canvas control in the dialog, you can go on to create the rectangle, line and graphictext controls in it.

Note: Graphictext controls do not repaint the background of the rectangle in which they are located. The background of the rectangle is specified at creation time of the graphictext control. What they do repaint is only the text specified in the text attribute.

To create dialog elements on a canvas control

Use the PROCESS GUI statement action ADD.

The rectangle, line and graphictext controls are then displayed inside the borders of the canvas control; if they exceed the canvas borders, they are clipped.

The following attributes are useful for controlling the behavior of the canvas control and the dialog elements on it:

- OFFSET-X and OFFSET-Y determine the x and y axis offset of the canvas control’s upper border against the upper border of the area by which the rectangle, line or graphictext control have exceeded the canvas control’s borders.
- RECTANGLE-X, RECTANGLE-Y, RECTANGLE-W and RECTANGLE-H determine the size of a rectangle control and its position relative to the underlying canvas control.
- P1-X, P1-Y, P2-X and P2-Y determine the start position (P1*xx*) and the end position (P2*xx*) of a line control relative to the underlying canvas control.

The following example illustrates how to create a canvas control and how to add four rectangle controls and two line controls dynamically (these could be used as a bar chart).

Example:

```
/* In the dialog's local data area, the following must be defined:
01 #CNV1 HANDLE OF CANVAS
01 #XAX HANDLE OF LINE
01 #YAX HANDLE OF LINE
01 #H1 HANDLE OF RECTANGLE
01 #H2 HANDLE OF RECTANGLE
01 #H3 HANDLE OF RECTANGLE
01 #H4 HANDLE OF RECTANGLE
01 #RESPONSE (I4)
/* In the dialog's AFTER-OPEN event handler, the following must be defined:
PROCESS GUI ACTION ADD WITH
PARAMETERS
  PARENT = #DLG$WINDOW
  TYPE = CANVAS
  HANDLE-VARIABLE = #CNV1
  RECTANGLE-X = 20
  RECTANGLE-Y = 20
  RECTANGLE-W = 200
  RECTANGLE-H = 200
  STYLE = 'F'
END-PARAMETERS
GIVING RESPONSE
PROCESS GUI ACTION ADD WITH
PARAMETERS
  PARENT = #CNV1
  TYPE = LINE
  HANDLE-VARIABLE = #YAX
  STYLE = 'S'
  P1-X = 20
  P1-Y = 20
  P2-X = 20
  P2-Y = 180
END-PARAMETERS
GIVING RESPONSE
PROCESS GUI ACTION ADD WITH
PARAMETERS
  PARENT = #CNV1
  TYPE = LINE
  HANDLE-VARIABLE = #XAX
  P1-X = 180
  P1-Y = 180
  P2-X = 20
  P2-Y = 180
END-PARAMETERS
GIVING RESPONSE
PROCESS GUI ACTION ADD WITH
PARAMETERS
  PARENT = #CNV1
  TYPE = RECTANGLE
  HANDLE-VARIABLE = #H1
  RECTANGLE-X = 20
  RECTANGLE-Y = 180
  RECTANGLE-H = 20
  RECTANGLE-W = -60
  FOREGROUND-COLOUR-NAME = BLACK
  BACKGROUND-COLOUR-NAME = RED
END-PARAMETERS
GIVING RESPONSE
PROCESS GUI ACTION ADD WITH
PARAMETERS
  PARENT = #CNV1
  TYPE = RECTANGLE
```

```
  HANDLE-VARIABLE = #H2
  RECTANGLE-X = 40
  RECTANGLE-Y = 180
  RECTANGLE-H = 20
  RECTANGLE-W = -40
  FOREGROUND-COLOUR-NAME = BLACK
  BACKGROUND-COLOUR-NAME = BLUE
END-PARAMETERS
GIVING RESPONSE
PROCESS GUI ACTION ADD WITH PARAMETERS
  PARENT = #CNV1
  TYPE = RECTANGLE
  HANDLE-VARIABLE = #H3
  RECTANGLE-X = 60
  RECTANGLE-Y = 180
  RECTANGLE-H = 20
  RECTANGLE-W = -55
  FOREGROUND-COLOUR-NAME = BLACK
  BACKGROUND-COLOUR-NAME = GREEN
END-PARAMETERS
GIVING RESPONSE
PROCESS GUI ACTION ADD WITH
PARAMETERS
  PARENT = #CNV1
  TYPE = RECTANGLE
  HANDLE-VARIABLE = #H4
  RECTANGLE-X = 80
  RECTANGLE-Y = 180
  RECTANGLE-H = 20
  RECTANGLE-W = -80
  FOREGROUND-COLOUR-NAME = BLACK
  BACKGROUND-COLOUR-NAME = MAGENTA
END-PARAMETERS
GIVING RESPONSE
```

Working with ActiveX Controls

ActiveX controls are third-party custom controls that you can integrate in a Natural dialog.

Terminology

ActiveX controls and Natural use different terminology in two cases:

ActiveX Control	Natural
Property	Attribute
Method	PROCESS GUI Statement Action

How To Define an ActiveX Control

The handle of an ActiveX Control is defined similar as a built-in dialog element, but its individual aspects are coded in double angle brackets.

Example:

```
01    #OCX-1 HANDLE OF <<OCX-Table.TableCtrl.1 [Table Control]>>
```

In the above example, ’Table.TableCtrl.1’ is the program ID (ProgID) under which the ActiveX control is registered in the system registry. The prefix ’OCX-’ identifies the control as an ActiveX control. ’[Table Control]’ is an optional part of the definition and provides a readable name.

How To Create an ActiveX Control

You create an instance of an ActiveX control by using the PROCESS GUI statement action ADD. To do so, the value of the TYPE attribute must be the ActiveX control’s ProgID prefixed with the string ’OCX-’ and put in double angle brackets. The ProgID is the name under which the control is registered in the system registry. You can additionally provide a readable name in square brackets. In addition to that, you can set Natural attributes such as RECTANGLE-X as well as the ActiveX control’s properties. The property name must be preceded by the string ’PROPERTY-’ and this combination must be put in double angle brackets. Furthermore, you can suppress the ActiveX control’s events. To do this, the event name must be preceded by the string ’SUPPRESS-EVENT’ this combination must be delimited by double angle brackets. The value of the SUPPRESS-EVENT property is either the Natural keyword ’SUPPRESSED’ or ’NOT-SUPPRESSED’.

Example:

```
PROCESS GUI ACTION ADD
    WITH PARAMETERS
       HANDLE-VARIABLE = #OCX-1
       TYPE = <<OCX-Table.TableCtrl.1 [Table Control]>>
       PARENT = #DLG$WINDOW
       RECTANGLE-X = 44
       RECTANGLE-Y = 31
       RECTANGLE-W = 103
       RECTANGLE-H = 46
       <<PROPERTY-HeaderColor>> = H'FF0080'
       <<PROPERTY-Rows>> = 16
       <<PROPERTY-Columns>> = 4
       <<SUPPRESS-EVENT-RowMoved>> = SUPPRESSED
       <<SUPPRESS-EVENT-ColMoved>> = SUPPRESSED
    END-PARAMETERS
```

Accessing Simple Properties

Simple properties are properties that do not have parameters. Simple properties of an ActiveX control are addressed like attributes of built-in controls. The attribute name is built by prefixing the property name with ’PROPERTY-’ and enclosing it in angle brackets.The properties of an ActiveX control are displayed in the Component Browser. The following examples assume that the ActiveX control #OCX-1 has the simple properties ’CurrentRow’ and ’CurrentCol’.

Example:

```
 * Get the value of a property.
 #MYROW := #OCX-1.<&ltPROPERTY-CurrentRow>>
 * Put the value of a property.
 #OCX-1.<&ltPROPERTY-CurrentCol>> := 17
```

The data types of ActiveX control properties are those defined by OLE Automation. In Natural, each of these data types is mapped to a corresponding Natural data type. The following table shows which OLE Automation data type is mapped to which Natural data type.

OLE Automation data type	NATURAL data type
VT_BOOL	L
VT_BSTR	A dynamic
VT_CY	P15.4
VT_DATE	T
VT_DECIMAL	Pn.m
VT_DISPATCH	HANDLE OF OBJECT
VT_ERROR	I4
VT_I1	I2
VT_I2	I2
VT_I4	I4
VT_INT	I4
VT_R4	F4
VT_R8	F8
VT_U1	B1
VT_U2	B2
VT_U4	B4
VT_UINT	B4
VT_UNKNOWN	HANDLE OF OBJECT
VT_VARIANT	(any Natural data type)
OLE_COLOR (VT_UI4)	B3
VT_FONT (VT_DISPATCH IFontDisp*)	HANDLE OF FONT, HANDLE OF OBJECT (IFontDisp*) A dynamic
VT_PICTURE (VT_DISPATCH IPictureDisp*)	HANDLE OF OBJECT (IPictureDisp*) A dynamic

Read the table in the following way: Assume an ActiveX control #OCX-1 has a property named ’Size’, which is of type VT_R8. Then the expression #OCX-1.<<PROPERTY-SIZE>> has the type F8 in Natural.

Note: The Component Browser displays the corresponding Natural data types directly.

Some special data types are considered individually in the following:

Colors

A property of type Color appears in Natural as a B3 value. The B3 value is interpreted as an RGB color value. The three bytes contain the red, green and blue elements of the color, respectively. Thus for example H’FF0000’ corresponds to red, H’00FF00’ corresponds to green, H’0000FF’ corresponds to blue and so on.

Example:

```
...
 01 #COLOR-RED (B3)
...
 #COLOR-RED := H'FF0000'
 #OCX-1.<<PROPERTY-BackColor>> := #COLOR-RED
...
```

Pictures

A property of type Picture appears in Natural as HANDLE OF OBJECT. Alternatively you can assign an Alpha value to a Picture property. The Alpha value must then contain the file name of a Bitmap (.bmp) file.

Example (usage of Picture properties):

```
...
 01 #MYPICTURE HANDLE OF OBJECT
...
 * Assign a Bitmap file name to a Picture property.
 #OCX-1.<<PROPERTY-Picture>>:= '11100102.bmp'
 *
 * Get it back as an object handle.
 #MYPICTURE := #OCX-1.<<PROPERTY-Picture>>
 *
 * Assign the object handle to a Picture property of another control.
 #OCX-2.<<PROPERTY-Picture>>:= #MYPICTURE
...
```

Fonts

A property of type Font appears in Natural as HANDLE OF OBJECT. You can alternatively assign a HANDLE OF FONT to a Font property. Additionally you can assign an Alpha value to a Font property. The Alpha value must then contain a font specification in the form that is returned by the STRING attribute of a HANDLE OF FONT.

Example 1 (using HANDLE OF OBJECT):

```
...
01 #MYFONT HANDLE OF OBJECT
...
* Create a Font object.
CREATE OBJECT #MYFONT OF CLASS ’StdFont’
#MYFONT.Name := ’Wingdings’
#MYFONT.Size := 20
#MYFONT.Bold := TRUE
*
* Assign the Font object as value to a Font property.
#OCX-1.<<PROPERTY-TitleFont>> := #MYFONT
...
```

Example 2 (using HANDLE OF FONT):

```
...
01 #FONT-TAHOMA-BOLD-2 HANDLE OF FONT
...
* Create a Font handle.
PROCESS GUI ACTION ADD WITH PARAMETERS
    HANDLE-VARIABLE = #FONT-TAHOMA-BOLD-2
    TYPE = FONT
    PARENT = #DLG$WINDOW
    STRING = ’/Tahoma/Bold/0 x -27/ANSI VARIABLE SWISS DRAFT/W/2/3/’
END-PARAMETERS GIVING *ERROR
...
* Assign the Font handle as value to a Font property.
#OCX-1.<<PROPERTY-TitleFont>> := #FONT-TAHOMA-BOLD-2
...
```

Example 3 (using a font specification string):

```
...
01 #FONT-TAHOMA-BOLD-2 HANDLE OF FONT
...
* Create a Font handle.
PROCESS GUI ACTION ADD WITH  PARAMETERS
    HANDLE-VARIABLE = #FONT-TAHOMA-BOLD-2
    TYPE = FONT
    PARENT = #DLG$WINDOW
    STRING = ’/Tahoma/Bold/0 x -27/ANSI VARIABLE SWISS DRAFT/W/2/3/’
END-PARAMETERS GIVING *ERROR
...
* Assign the font specification as value to a Font property.
#OCX-1.<<PROPERTY-TitleFont>> := #FONT-TAHOMA-BOLD-2.STRING
...
```

Variants

A property of type Variant is compatible with any Natural data type. This means that the type of the expression #OCX-1.<<PROPERTY-Value>> is not checked by the compiler, if "Value" is a property of type Variant. So the assignments #OCX-1.<<PROPERTY-Value >> := #MYVAL and #MYVAL := #OCX-1.<<PROPERTY-Value >> are allowed independently of the type of the variable #MYVAL. It is however up to the ActiveX control to accept or reject a particular property value at runtime, or to deliver the value in the requested format. If it does not, the ActiveX control will usually raise an exception. This exception is returned as a Natural error code to the Natural program. Here it can be handled in the usual way in an ON ERROR block. You should check the documentation of the ActiveX control to find out which data formats are actually allowed for a particular property of type Variant.

An expression like #OCX-1.<<PROPERTY-Value>> (where "Value" is a Variant property) can occur as source operand in any statement. However, it can be used as target operand only in assignment statements.

Examples (usage of Variant properties):
(Assume that ’Value’ is a property of type Variant of the ActiveX control #OCX-1)

```
...
01 #STR1 (A100)
01 #STR2 (A100)
...
* These statements are allowed, because the Variant property is used
* as source operand (its value is read).
#STR1 := #OCX-1.<<PROPERTY-Value>>
COMPRESS #OCX-1.<<PROPERTY-Value>> ’XYZ’ to #STR2
...
* This leads to an error at compiletime, because the Variant
* property is used as target operand (its value is modified) in
* a statement other than an assignment.
COMPRESS #STR1 "XYZ" to #OCX-1.<<PROPERTY-Value>>
...
* This statement is allowed, because the Variant property is used
* as target operand in an assignment.
COMPRESS #STR1 ’XYZ’ to #STR2
#OCX-1.<<PROPERTY-Value>> := #STR2
...
```

Arrays

A property of type SAFEARRAY of up to three dimensions appears in a Natural program as an array with the same dimension count, occurrence count per dimension and the corresponding format. (Properties of type SAFEARRAY with more than three dimensions cannot be used in Natural programs.) The dimension and occurrence count of an array property is not determined at compiletime but only at runtime. This is because this information is variable and is not defined at compiletime. The format however is checked at compiletime.

Array properties are always accessed as a whole. So no index notation is necessary and allowed with an array property.

Examples (usage of Array properties):
(Assume that ’Values’ is a property of the ActiveX control #OCX-1 an has the type SAFEARRAY of VT_I4)

```
...
01 #VAL-L (L/1:10)
01 #VAL-I (I4/1:10)
...
* This statement is allowed, because the format of the property
* is data transfer compatible with the format of the receiving array.
* However, if it turns out at runtime that the dimension count or
* occurrence count per dimension do not match, a runtime error will
* occur.
VAL-I(*) := #OCX-1.<<PROPERTY-Values>>
...
* This statement leads to an error at compiletime, because
* the format of the property is not data transfer compatible with
* the format of the receiving array.
VAL-L(*) := #OCX-1.<<PROPERTY-Values>>
...
```

Using The PROCESS GUI Statement

The methods of ActiveX controls are called as actions in a PROCESS GUI statement. The same is the case with the complex properties of ActiveX controls (i. e. properties that have parameters). The methods and properties of an ActiveX control are displayed in the Component Browser

Performing Methods

To perform a method of an ActiveX control the PROCESS GUI statement is used. The name of the corresponding PROCESS GUI action is built by prefixing the method name with ’METHOD-’ and enclosing it in angle brackets. The ActiveX control handle and the method parameters (if any) are passed in the WITH clause of the PROCESS GUI statement The return value of the method (if any) is received in the variable specified in the USING clause of the PROCESS GUI statement.

This means: To perform a method, you enter a statement

```
    PROCESS GUI ACTION <<METHOD-methodname>> WITH handlename [ parameter]...
[USING method-return-operand]..
```

Examples:

```
* Performing a method without parameters:
PROCESS GUI ACTION <<METHOD-AboutBox>> WITH #OCX-1
* Performing a method with parameters:
PROCESS GUI ACTION <<METHOD-CreateItem>> WITH #OCX-1 #ROW #COL #TEXT
* Performing a method with parameters and a return value:
PROCESS GUI ACTION <<METHOD-RemoveItem>> WITH #OCX-1 #ROW #COL USING #RETURN
```

Formats and length of the method parameters and the return value are checked at compiletime against the definition of the method, as it is displayed in the Component Browser

Getting Property Values

To get the value of a property that has parameters, the name of the corresponding PROCESS GUI action is built by prefixing the property name with ’GET-PROPERTY-’ and enclosing it in angle brackets. The ActiveX control handle and the property parameters (if any) are passed in the WITH clause of the PROCESS GUI statement The property value is received in the USING clause of the PROCESS GUI statement.
This means:
To get the value of a property that has parameters, you enter a statement

```
    PROCESS GUI ACTION <<GET-PROPERTY-propertyname>> WITH handlename [ parameter]
... USING get-property-operand
```

Example:

```
PROCESS GUI ACTION <<GET-PROPERTY-ItemHeight>> WITH #OCX-1 #ROW #COL USING #ITEMHEIGHT
```

Formats and length of the property parameters and the property value are checked at compiletime against the definition of the method, as it is displayed in the Component Browser

Putting Property Values

To put the value of a property that has parameters, the name of the corresponding PROCESS GUI action is built by prefixing the property name with ’PUT-PROPERTY-’ and enclosing it in angle brackets. The ActiveX control handle and the property parameters (if any) are passed in the WITH clause of the PROCESS GUI statement The property value is passed in the USING clause of the PROCESS GUI statement.

This means:
To put the value of a property that has parameters, you enter a statement

```
    PROCESS GUI ACTION <<PUT-PROPERTY-propertyname>> WITH handlename [ parameter]
... USING put-property-operand
```

Example:

```
    PROCESS GUI ACTION <<PUT-PROPERTY-ItemHeight>> WITH #OCX-1 #ROW #COL USING #ITEMHEIGHT
```

Formats and length of the property parameters and the property value are checked at compiletime against the definition of the method, as it is displayed in the Component Browser

Optional Parameters

Methods of ActiveX controls can have optional parameters. This is also true for parameterized properties. Optional parameters need not to be specified when the method is called. To omit an optional parameter, use the placeholder 1X in the PROCESS GUI statement. To omit n optional parameters, use the placeholder nX.

In the following example it is assumed that the method SetAddress of the ActiveX control #OCX-1 has the parameters FirstName, MiddleInitial, LastName, Street and City, where MiddleInitial, Street and City are optional. Then the following statements are correct:

Example:

```
* Specifying all parameters.
PROCESS GUI ACTION <<METHOD-SetAddress>> WITH #OCX-1
FirstName MiddleInitial LastName Street City
* Omitting one optional parameter.
PROCESS GUI ACTION <<METHOD-SetAddress>> WITH #OCX-1
FirstName 1X LastName Street City
* Omitting the optional parameters at end explicitly.
PROCESS GUI ACTION <<METHOD-SetAddress>> WITH #OCX-1
FirstName MiddleInitial LastName 2X
* Omitting the optional parameters at end implicitly.
PROCESS GUI ACTION <<METHOD-SetAddress>> WITH #OCX-1
FirstName MiddleInitial LastName
```

Omitting a non-optional (mandatory) parameter results in a syntax error.

Error handling

The GIVING clause of the PROCESS GUI statement can be used as usual to handle error conditions. The error code can either be caught in a user variable and then be handled, or the normal Natural error handling can be triggered and the error condition be handled in an ON ERROR block.

Example:

```
DEFINE DATA LOCAL
1 #RESULT-CODE (N7)
...
END-DEFINE
...
* Catching the error code in a user variable:
PROCESS GUI ACTION <<METHOD-RemoveItem>> WITH #OCX-1 #ROW #COL USING #RETURN GIVING #RESULT-CODE
*
* Triggering the Natural error handling:
PROCESS GUI ACTION <<METHOD-RemoveItem>> WITH #OCX-1 #ROW #COL USING #RETURN GIVING *ERROR-NR
...
```

Special error conditions that can occur during the execution of ActiveX control methods are:

- A method parameter, method return value or property value could not be converted to the data format expected by the ActiveX control. (These format checks are normally already done at compiletime. In these cases no runtime error can be expected. However, note that method parameters, method return values or property values defined as Variant are not checked at compiletime. This applies also for arrays and for those data types that can be mapped to several possible Natural data types.)
- A COM or Automation error occurs while locating and executing a method.
- The ActiveX control raises an exception during the execution of a method.

In these cases the error message contains further information provided by the ActiveX control, which can be used to determine the reason of the error with the help of the documentation of the ActiveX control.

Using Events With Parameters

Events sent by ActiveX controls can have parameters. In the controls event-handler sections, these parameters can be queried. Parameters passed by reference can also be modified. The events of an ActiveX control, the names and data types of the parameters and the fact if a parameter is passed by value or by reference is all displayed in the Component Browser.

Event parameters of an ActiveX control are addressed like attributes of built-in controls. The attribute name is built by prefixing the parameter name with 'PARAMETER-' and enclosing it in angle brackets. Alternatively, parameters can be addressed by position. This means the attribute name is built by prefixing the number of the parameter with 'PARAMETER-' and enclosing it in angle brackets.The first parameter of an event has the number 1, the second the number 2 and so on. These attribute names are only valid inside the event handler of that particular event.

In the following examples it is assumed that a particular event of the ActiveX control #OCX-1 has the parameters KeyCode and Cancel. Then the event handler of that event might contain the following statements:

Example:

```
* Querying a parameter by name:
#PRESSEDKEY := #OCX-1.<<PARAMETER-KeyCode>>
* Querying a parameter by position:
#PRESSEDKEY := #OCX-1.<<PARAMETER-1>>
```

Parameters that are passed by reference can be modified in the event handler. In the following example it is assumed that the Cancel parameter is passed by reference and is thus modifiable. Then the event handler might contain the following statements:

Example:

```
* Modifying a parameter by name:
#OCX-1.<<PARAMETER-Cancel>>:= TRUE
* Modifying a parameter by position:
#OCX-1.<<PARAMETER-2>>:= TRUE
```

Suppressing Events At Runtime

To suppress or unsuppress an event of an ActiveX control at runtime, modify the corresponding suppress event attribute of the control. The name of the suppress event attribute is built by prefixing the event name with 'SUPPRESS-EVENT-' and enclosing it in angle brackets. The events of an ActiveX control are displayed in the Component Browser.

The following example assumes that the ActiveX control #OCX-1 has the event ColMoved.

Example:

```
* Suppress the event.
#OCX-1.<<SUPPRESS-EVENT-ColMoved>> := SUPPRESSED
* Unsuppress the event.
#OCX-1.<<SUPPRESS-EVENT-ColMoved>> := NOT-SUPPRESSED
```

Working with Arrays of Dialog Elements

It is sometimes convenient to arrange dialog elements in one or two dimensions. If, for example, you want to arrange several radio-button controls in one column, it is possible to draw the first one and specify the others as a one-dimensional array.

To work with arrays of dialog elements:

1. Click the "Array" button in the radio-button control’s attributes window.
   The "Array Specification" dialog box appears.
2. Enter:

- the number of dimensions;
- the bounds of the first and second dimension, if applicable;
- the spacing on the x and y axis in pixels (depending on whether the array is arranged in rows or in columns);
- the arrangement (rows or columns).

The array will now be treated as a graphical entity. Note that you will have to assign a common GROUP-ID attribute to each radio-button control. This will enable you to treat the array as a logical entity.

For each dialog element in an array, the following attributes may be specified separately:

- STRING
- DIL-TEXT
- BITMAP-FILE-NAME

In an event handler for an array of dialog elements, the system variable *CONTROL will denote one of the array elements.

If a variable is selected as the source of an attribute value, the array must contain at least the index ranges of the dialog element.

If a message file ID is specified as the source of an attribute value, consecutive messages are taken for the array’s sequence of dialog elements.

In an array of dialog elements, you can assign one value to all dialog elements in the array using the (*) notation or a range, such as in the following examples:

Examples:

```
#PB-1.ENABLED(*) := TRUE    /*invalid
#PB-1.ENABLED(1:3) := TRUE  /*invalid
```

An alternative way of creating a sequence of identical dialog elements is to duplicate or copy and paste an individual dialog element and use the grid plus the cross-hair cursor to place them.

The following example illustrates how to set the STRING attribute of occurence 2 in a one-dimensional push-button array:

Examples:

```
#PB-2.STRING(2) := ’HUGO’
```

Working with Control Boxes

A control box is is used to enhance the effectiveness of the nested control support. However, control boxes have a number of unique features that merit their separate discussion.

Control boxes are, in themselves, fairly inert controls, belonging to the same category as text constants and group frames in that they cannot receive the focus and do not receive any mouse or keyboard input. Instead, they are intended to act as general-purpose containers for other controls (including, possibly, other control boxes), in order to build up a control hierarchy. In doing so, control boxes support three styles which are worthy of special mention here:

- Because it is often desirable to be able to group controls together for convenience, but not desirable that the user actually sees the container itself, control boxes can be marked with the style ’transparent’. In this case, no parts of the control box are drawn, and any underlying colors and controls show through.
- Control boxes can also be marked with the style ’exclusive’. When an exclusive control box is made visible, either in the dialog editor or at runtime, all other sibling control boxes that are also marked as ’exclusive’ are hidden. This applies to edit-time and runtime in a slightly different way. At runtime, setting the VISIBLE attribute of an exclusive control box to TRUE hides all its exclusive siblings and sets their VISIBLE attribute to FALSE. At edit-time, whenever an exclusive control box or one of its descendants is selected, the exclusive control box becomes visible and all other exclusive siblings are hidden. However, in this latter case the VISIBLE attribute of the controls concerned is unaffected. This implies that the exclusive control box that is initially visible when the dialog is run is independent of the exclusive control box that was visible at the time the dialog was last saved.
- Additionally, control boxes support the ’size to parent’ style. When a container control, or the dialog itself, is resized, all child control boxes (if any) with this style set are resized to entirely fill the parent’s client area. The same applies when this style is first set in the dialog editor. However, it is still possible to resize such control boxes independently of their container.

Purpose of exclusive control boxes

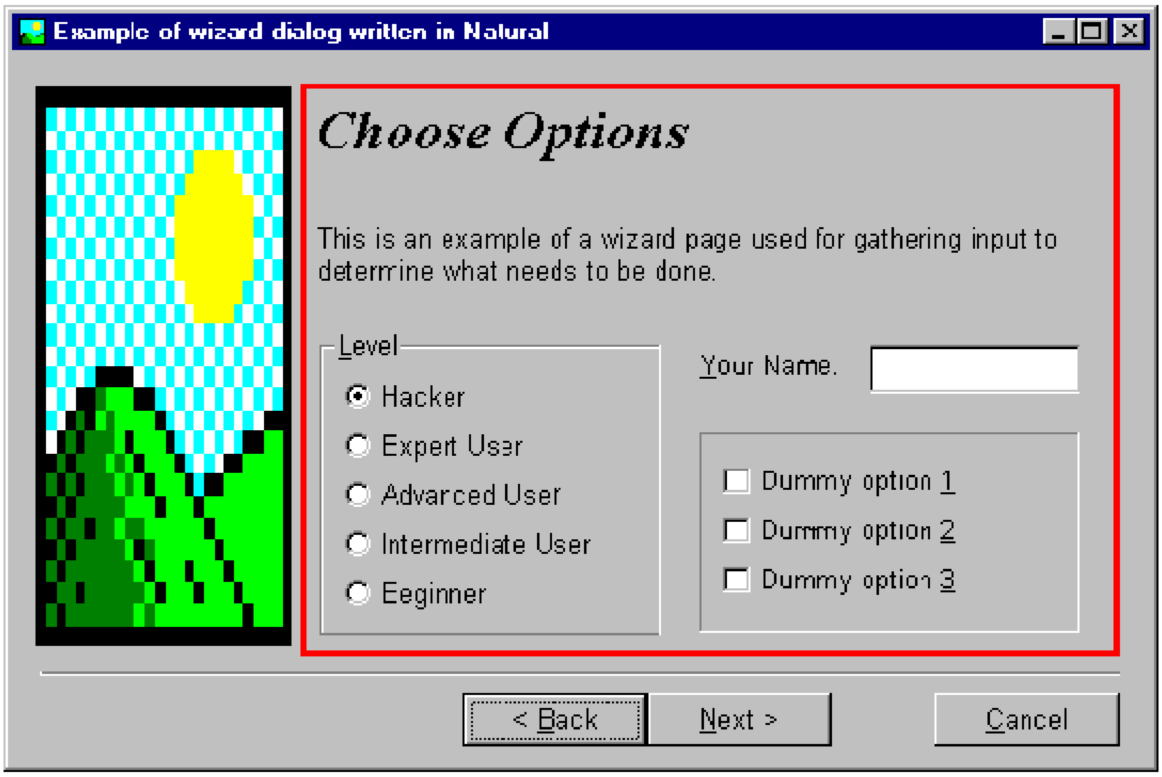
Exclusive control boxes, as described above, are primarily intended for situations where it is necessary to manage several overlapping "pages" of controls occupying the same region of a dialog. Without the auto-hiding feature which exclusive control boxes provide, it would be very difficult indeed for a user to handle this situation in the dialog editor, as many controls would be partially or completely overlapped by others. Of course, one could move the control to the front of the control sequence during editing, but this would be highly inconvenient, and one would have to remember to move the control back before continuing.

Using exclusive control boxes, editing a control in this situation is as simple as selecting it. For controls that are not currently on display, the selection can be made via the combo box in the dialog editor’s status bar or by using the <Tab> key to walk through the controls sequentially until the target control is reached. When a control that is a descendant of an exclusive control box is selected, that exclusive control box is made visible (if not already so), and the previously visible exclusive control box is hidden. These changes have no impact on the generated dialog source code and the runtime state of the dialog.

Examples of use of exclusive control boxes

Although the design of control boxes was intended to keep them as general as possible, two possible situations where overlapping control pages are desired (and hence where exclusive control boxes become extremely useful) are worthy of special mention here:

- Wizard dialogs.
- Tabbed dialogs ("Property sheets").



Example of wizard dialog written in Natural

Choose Options

This is an example of a wizard page used for gathering input to determine what needs to be done.

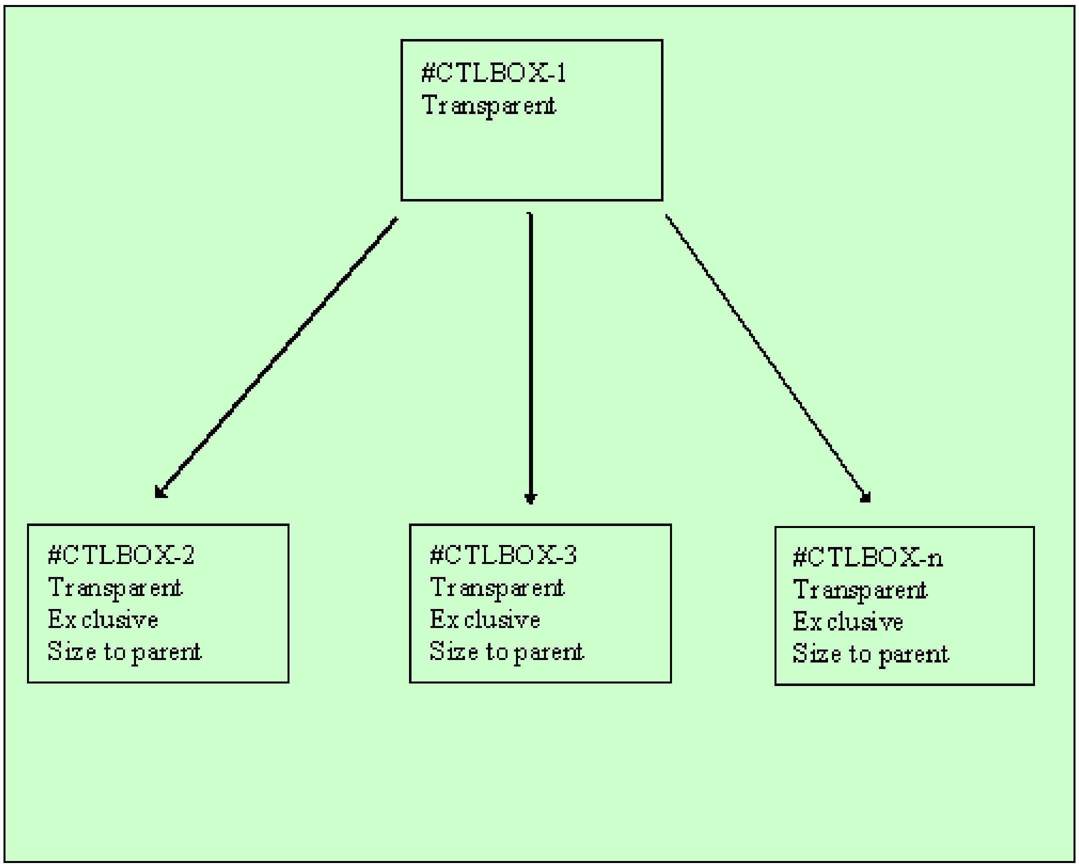
Level

[✓ Hacker]
Expert User
Advanced User
Intermediate User
Beginner

Your Name:

☐ Dummy option 1
☐ Dummy option 2
☐ Dummy option 3

Within the rectangle highlighted in red, the so-called "wizard pages" are displayed. Within this area, we use a 2-level hierarchy of control boxes in order to implement the required functionality:



Here, #CTLBOX-1 is used as the "master" control box, which makes resizing of the pages easier later, should this become necessary. Because all child control boxes are marked with the style ’size to parent’, we can resize the wizard page area simply by resizing #CTLBOX-1.

The child control boxes are used to implement the actual wizard pages. #CTLBOX-2 contains the controls used for wizard page 1, #CTLBOX-3 contains the controls for wizard page 2, and so on.

Creation of the wizard pages

Creation of the wizard pages typically involves the following steps:

1. Create the top-level ("master") control box as for any other control.
2. Via its attributes window, set the ’transparent’ style.
3. Create another control box within the first one. The new control box automatically becomes a child of the first one, because control boxes are always containers.
4. Via the attributes window for the child control box, set the ’transparent’, ’exclusive’ and ’size to parent styles’. Because the ’size to parent’ style is set, the child control box expands to fill its container.
5. Now you can start adding the controls onto the newly-created control box, which becomes wizard page 1.
6. Adding a new wizard page is most easily achieved by selecting the child control box you wish to immediately precede the new one, then using the clipboard copy and paste commands. Before doing the copy, Natural will prompt you as to whether you want the child controls to be copied, too. Answer this question with ’No’.
7. Because the newly added child control box also has the exclusive flag set, the previously displayed child control box is hidden, and the new blank one is shown, ready for you to start adding a new set of controls as for the first wizard page.

Switching between the wizard pages at edit-time

Switching between the pages at edit time can be most simply achieved by selecting the child control box for the appropriate page, or one of the controls on it, from the combo box in the dialog editor’s status bar.

Creating the divider line

The divider line between the push buttons and the wizard pages can be implemented as a very thin group box (2 pixels high) with no caption. The still slightly visible sides of the group box at each end can be masked out by using a transparent control box which comes after the group frame in the control sequence. Make sure the ’control clipping’ style for the dialog is switched on for this technique to work.

Implementing the ’Back’ and ’Next’ push buttons

Firstly, define a local variable for the dialog to store the handle of the currently active page. E.g.:

```
 01 #ACTPAGE HANDLE OF CONTROLBOX ...
```

Secondly, set this variable to the handle of the first wizard page in the AFTER-OPEN event for the dialog:

```
 #ACTPAGE := #CTLBOX-1.FIRST-CHILD ..
```

where #CTLBOX-1 is the handle of the top-level control box.
Now we are ready to implement the CLICK event code for the ’Next’ push button (#PB-NEXT). This could look something like this:

```
IF #ACTPAGE.SUCCESSOR = NULL-HANDLE
  CLOSE DIALOG *DIALOG-ID
ELSE
  REPEAT
    #ACTPAGE := #ACTPAGE.SUCCESSOR
    WHILE #ACTPAGE.ENABLED = FALSE
  END-REPEAT
  #ACTPAGE.VISIBLE := TRUE
  IF #ACTPAGE.SUCCESSOR = NULL-HANDLE
    #PB-NEXT.STRING := ’Finish’
    #PB-BACK.ENABLED := FALSE
    #PB-CANCEL.ENABLED := FALSE
  ELSE
    #PB-BACK.ENABLED := TRUE
  END-IF
END-IF
..
```

Note that this logic does not be modified if further wizard pages are added later. Note also that any intermediate wizard pages whose corresponding control box has been disabled are ignored. This allows certain wizard pages to be skipped, based on previous input, by simply setting the relevant control box ENABLED attribute to FALSE. When the last page is reached, the text for the ’Next’ push button is changed to ’Finish’.

The CLICK event code for the ’Back’ push button (#PB-BACK) is very similar:

```
REPEAT
  #ACTPAGE := #ACTPAGE.PREDECESSOR
  WHILE #ACTPAGE.ENABLED = FALSE
END-REPEAT
IF #ACTPAGE.PREDECESSOR = NULL-HANDLE
  #PB-BACK.ENABLED := FALSE
END-IF
#ACTPAGE.VISIBLE := TRUE
..
```

Note that the ’Back’ push button should be initially disabled in the dialog editor.

Clearing all controls on a wizard page

This can be conveniently achieved by selecting any (highest-level) control on the relevant page, then performing a "Select All" from the "Edit" menu to additionally select all the controls siblings. The selected controls can then be deleted as normal.

Example 2 - a tabbed dialog

A tabbed dialog (sometimes called a "property sheet") is very similar in concept to a wizard dialog. The only substantial difference is that instead of navigating between the control "pages" via the ’Next’ and ’Back’ push buttons, the user directly accesses the page he wants by clicking on the appropriate tab. The control page hierarchy can be built up and handled in the dialog editor in the same way as in the wizard dialog example above. Several ActiveX controls are available which provide the actual tabs.

It should be noted, however, that the switching between the pages (i.e., switching between the corresponding control boxes) is not automatic. The Natural programmer must insert code for the ActiveX event raised by a tab switch, find out which tab is selected, and set the VISIBLE attribute of the appropriate (exclusive) control box to TRUE. This cannot be done implicitly by Natural because each ActiveX control can implement its functionality in any way it chooses. There is no standard event raised for a tab switch and no standard method with standard parameters (or standard property) for determining the currently active tab.

An example tabbed dialog, making use of the Microsoft "Tab Strip" ActiveX control (V4-NEST.NS3) is shipped as part of the Natural example libraries.

Working with Error Events

When a runtime error occurs while a dialog is active, the dialog receives an error event. You can specify event-handler code to be executed whenever this error occurs. If no error event-handler code is specified, Natural aborts with an error message and all dialogs will be closed.

You can continue normal dialog processing after error handling by specifying ESCAPE ROUTINE at the end of the event-handler code.

The dialog editor generates an ON ERROR statement for the event handler. If, for example, you want to prevent the end user from closing the entire application when trying to divide an integer by zero, and the parameter ZD is set to ON, the error event-handler code might look like this:

```
COMPRESS 'Natural error' *ERROR 'occurred.' INTO #DLG$WINDOW.STATUS-TEXT
    ESCAPE ROUTINE
```

Working with a Group of Radio-Button Controls

radio-button controls are created just like push-button controls or toggle button controls; however, they are grouped using the GROUP-ID attribute. If you define a number of radio-button controls as a group, only one button is selected at any time. The GROUP-ID attribute provides this selection logic.

You group several radio-button controls by assigning them the same GROUP-ID value (group number) in their attributes windows. If the end user clicks on a radio-button control, all other radio-button controls in the dialog with the same GROUP-ID will be deselected. They will also be deselected if one radio-button control is selected by code like the following:

Example:

```
...
   1 #RB-1 HANDLE OF RADIOBUTTON
...
#RB-1.CHECKED := CHECKED  /* Set the CHECKED attribute to value CHECKED
...
```

You also have to bear in mind that the end user should be able to use the keyboard for navigation inside a group of radio-button controls: TAB selects the first radio-button control, and the arrow keys enable you to navigate within the radio-button group. To ensure that Natural automatically allows for such navigation, the radio-button controls must follow each other directly in the navigation sequence. If you are dynamically adding a radio-button control via the PROCESS GUI statement action ADD, this can be achieved by specifying a value for the button's FOLLOWS attribute.

To edit the navigation sequence

From the menu bar, select "Dialog > Control Sequence".

Working with List-Box Controls and Selection-Box Controls

list-box controls and selection box controls contain a number of items. Both the controls and the items are dialog elements; the controls are the parents of the items.

There are two ways of creating list-box items and sebox items:

- Use Natural code to create individual and multiple list-box items dynamically; or
- use the dialog editor (to add single or arrays of list-box items and sebox items).

In Natural code, this may look like this:

Example:

```
#AMOUNT := 5
ITEM (1) := 'BERLIN'
ITEM (2) := 'PARIS'
ITEM (3) := 'LONDON'
ITEM (4) := 'MILAN'
ITEM (5) := 'MADRID'
PROCESS GUI ACTION ADD-ITEMS WITH #LB-1 #AMOUNT #ITEM (1:5) GIVING #RESPONSE
```

You first specify the number of items you want to create, name the items, and use the PROCESS GUI statement action ADD-ITEMS.

If you want to go through all items of a list-box control to find out which ones are selected, it is advisable to use the SELECTED-SUCCESSOR attribute because if a list-box control contains a large number of items (100, for example), this helps improve performance. If you use SELECTED-SUCCESSOR, you have one query instead of 100 individual queries if you use the attributes SELECTED and SUCCESSOR.

Example:

```
/* Displays the STRING attribute of every SELECTED list-box item
MOVE #LISTBOX.SELECTED-SUCCESSOR TO #LBITEM
REPEAT UNTIL #LBITEM = NULL-HANDLE
   .../* STRING display logic
   MOVE #LBITEM.SELECTED-SUCCESSOR TO #LBITEM
END-REPEAT
```

For performance reasons, you should not use the SELECTED-SUCCESSOR attribute to refer to the same dialog element handle twice, because Natural goes through the list of item handles twice:

Example:

```
/* Displays the STRING attribute of every SELECTED list-box item,
/* but may be slow
MOVE #LISTBOX.SELECTED-SUCCESSOR TO #LBITEM
REPEAT UNTIL #LBITEM = NULL-HANDLE
   IF #LBITEM.SELECTED-SUCCESSOR = NULL-HANDLE /* Searches in the list of items
     IGNORE
   END-IF
   .../* STRING display logic
   MOVE #LBITEM.SELECTED-SUCCESSOR TO #LBITEM /* Searches in the list of items
END-REPEAT                                    /* for the second time
```

To avoid this problem, you use a second variable "#OLDITEM" besides "#LBITEM":

Example:

```
/* Displays the STRING attribute of every SELECTED list-box item
MOVE #LISTBOX.SELECTED-SUCCESSOR TO #LBITEM
REPEAT UNTIL #LBITEM = NULL-HANDLE
   #OLDITEM = #LBITEM
   #LBITEM = #LBITEM.SELECTED-SUCCESSOR/* Searches in the list of items (once)
   IF #LBITEM = NULL-HANDLE
     IGNORE
  END-IF
   .../* Display logic using #OLDITEM.STRING
END-REPEAT
```

If you retrieve the handle values of the selected items, a value other than NULL-HANDLE would normally be returned by selected items. Such a handle value can also be returned by non-selected items if you assign SELECTED-SUCCESSOR a value immediately before retrieving the SELECTED-SUCCESSOR value of a non-selected item, as shown in the following example:

Example:

```
...
PTR := #LB-1.SELECTED-SUCCESSOR
PTR := NOT_SELECTEDHANDLE.SELECTED-SUCCESSOR
IF NOT_SELECTEDHANDLE.SELECTED-SUCCESSOR = NULL-HANDLE THEN
   #DLG$WINDOW.STATUS-TEXT := 'NULL-HANDLE'
ELSE
   COMPRESS 'NEXT SELECTION: ' PTR.STRING TO #DLG$WINDOW.STATUS-TEXT
END-IF
...
```

If you want to query whether a particular item in a list-box control is selected, you get the best performance by using the SELECTED attribute:

Example:

```
#DLG$WINDOW.STRING:= #LB-1-ITEMS.SELECTED(3)
```

Protecting Selection-Box Controls and Input-Field Controls

To prevent an end user from typing in input data in a sebox control or input-field control, you have several possiblities, for example:

- setting the MODIFIABLE attribute to FALSE for the dialog element, or
- setting session parameter AD=P, or
- using a control variable (CV).

If a sebox control is protected, it is still possible to select items; only values from the item list will be displayed in its input field. If the STRING attribute is set to a value (dynamically or by initialisation) which is not in the item list, the value will not be visible to the end user.

Working with Nested Controls

It is possible to create controls as children of other controls in addition to so-called "top-level" controls, which are direct children of the dialog. Such controls are referred to as nested controls. The parent control is referred to as the container. We will also use the term siblings to refer to a set of child controls which all have the same parent. Clearly, there can be many different sets of sibling controls within a control hierarchy.

Creation of a control hierarchy enables the Natural programmer to group together controls such that they can be manipulated more easily and more efficiently within a Natural program. The following list describes the characteristics of nested controls:

- Their position is relative to the client area of the container control instead of relative to the dialog.
- Their display is clipped to their respective ancestor windows. This means that the areas of the nested control that are outside the boundary of its container are not visible. The dialog editor does not allow dragging of nested controls outside of the container.
- Nested controls are always displayed in front of their container control, regardless of their position in the control sequence.
- Nested controls are moved with their container control. This applies at both edit-time in the dialog editor (when the container is dragged) and at runtime (when the container's RECTANGLE-X and/or RECTANGLE-Y attributes are modified).
- Nested controls are hidden when the container control is hidden, even though the VISIBLE attribute of the nested control remains unchanged.
- Nested controls are disabled at runtime when the container control is disabled, even though the ENABLED attribute of the nested control remains unchanged and even though the control does not become grayed.
- Nested controls are deleted when the container control is deleted.

Note:
Natural does not impose any arbitrary limits on the number of levels that a control hierarchy may contain. The level number for a particular control is displayed together with the control's name in the dialog editor status bar combo box.

Which control types can be containers?

Not all control types are capable of acting as a container. It is not possible to create a control as a child of an input field, for example. There are currently three types of container control supported by Natural:

- Group frames that have the (new) 'container' style set. This can be changed in the dialog editor (via its attributes window) after the group frame has been created. If a group frame is converted to a container, all controls that are spatially contained within it are moved in the control hierarchy to become descendants of the group frame. If a group frame is converted to a non-container, all direct children of the group frame are moved up a level in the hierarchy to become siblings of the group frame.
- ActiveX controls which are marked as "OLEMISC_SIMPLEFRAME" in the registry. This flag is fixed by design for a particular ActiveX control class.
- Control boxes. This control type is always a control container. Indeed, that is its entire purpose in life. See the section "Working with Control Boxes" for more information.

Creating a nested control

Nested controls are created in the dialog editor in the same way as non-nested controls are. If, during control insertion, the initial left mouse button click is determined to be over a container control, the new control is created automatically by Natural as a child of that container. Even before the mouse button is clicked in insert mode, the dialog editor’s status bar is continually updated with the container-relative mouse co-ordinates as the mouse cursor traverses the dialog.

In addition, nested controls can be indirectly created within the dialog editor when converting group frames to containers as described above.

At runtime, nested controls can be created dynamically, via the PROCESS GUI ACTION ADD statement for the nested control, by specifying the PARENT attribute as the handle of the required container control instead of the handle of the dialog. The nested control’s position (RECTANGLE-X and RECTANGLE-Y attributes) should be specified relative to the container’s client area. The client area of a control is the internal area of a control, excluding frame components such as 3-D borders, single-pixel frames resulting from used of the ’Framed’ style, and a control’s scroll bars.

Multiple selection, control sequence and clipboard operations

The dialog editor prohibits selection of multiple controls which do not have the same parent (i.e., are not all siblings of each other). This applies regardless of whether multiple controls are selected via "rubber banding" (marking of a region with the left mouse button held down) or via extended selection (holding down the <Shift> key whilst selecting a control). However, if a selected container control is deleted, then all its direct and indirect children (*descendants*) are of course implicitly deleted also, even though they are not explicitly selected. For this reason, a clipboard cut operation always copies the selected control(s) AND all descendant controls (if any) to the clipboard. For a clipboard copy operation, it is not clear whether to copy the container alone, or the container plus all its descendants. In this case, a message box is displayed, allowing the user to choose between the two options.

The pasting of controls from the clipboard uses the same control sequence (tab order) insertion position logic as for a control created from scratch. In both cases, the new control is created at a position in the control sequence immediately following the selected sibling (if any) plus any of its successive descendants. If a control other than a sibling is selected, an "effective sibling" is used instead, based on the position of the (active) selected control in the control sequence. The "active" selected control is the selected control (if any) which is highlighted using black (rather than gray) selection handles. If no selection is active, the control is inserted into the control sequence immediately preceding the first sibling control, or immediately after its container (or at the front of the control sequence for top-level controls) if the container is empty. Note, however, that the control sequence is maintained independently of the hierarchy. After a control has been created, it is possible to explicitly move any control to any position in the control sequence via the Control Sequence option on the Dialog menu.

The position of the newly-created control in the hierarchy is determined slightly differently in these two cases. In the case of a control being created from scratch, the container is determined by searching for the (topmost) container at the position where the left mouse button was pressed. However, in the case of pasting from the clipboard, we have no (X, Y)-position which we can use. In this case, the container is assumed to be the container of the selected control(s), or the dialog itself if no controls are selected. This means that if, for example, it is desired to copy and paste a control from one container to another, a control within the second container must be selected prior to the paste, not the container itself. If the second container is empty, this requires temporary creation of a dummy child control first, which can be deleted after the paste operation is complete.

Deletion of controls also deletes any of their descendant controls.

With the introduction of nested controls, the ’Select All’ command has been changed to operate in the following manner:

- If no control is currently selected, the command selects all top-level controls.
- Otherwise, all other controls that are siblings of the currently selected one(s) are additionally selected.

Thus, in the common case where only one level of hierarchy is in use, the ’Select All’ command continues, as before, to select all dialog controls.

Working with a Dynamic Information Line

Event-driven applications are much more user-friendly when text in the dynamic information line (DIL) explains the dialog element that currently has the focus. A dialog element has the focus if it can receive the end user’s keyboard input.

You have two options to relate a dialog element to a DIL text:

- Use the dialog editor (most likely because it is the easiest way); or
- use Natural code to specify everything dynamically.

When you use the dialog editor, you will have to go through the following steps:

1. Set the attribute HAS-DIL to TRUE for the dialog by marking the "Dyn. Info Line" entry in the "Dialog Attributes" window.
2. Set the attribute DIL-TEXT to ’*diltextstring*’ for the dialog element. Push the "Source..." button to the right of the "DIL Text:" entry in the attributes window. The window "Specify attribute Source" appears. Choose one of the attribute sources and enter the text in the "Value" field. Ensure that ’*diltextstring*’ explains the dialog element’s usage in a short phrase.

When you use Natural code, the above two steps may look like this:

Example:

```
...
PERSDATA-DIALOG.HAS-DIL := TRUE  /* Set HAS-DIL To TRUE
#PB-1.DIL-TEXT := ’DILTEXTSTRING’   /* Assign the text string
...
```

Note: The STATUS-TEXT and the DIL-TEXT are displayed in the same area if the dialog has a status line and a text is displayed on the DIL.

Working with a Status Bar

In a similar way as the dynamic information line, the status bar makes an event-driven application more user-friendly.

The programmer has two options to relate a dialog element to a status bar:

- use the dialog editor (most likely because it is the easiest way); and
- use Natural code to specify everything dynamically.

When you use the dialog editor, you will have to:

- Set the attribute HAS-STATUS-BAR to TRUE for the dialog by marking the "Status Bar" entry in the "Dialog Attributes" window. The HAS-STATUS-BAR attribute determines whether the status bar may be modified. If HAS-STATUS-BAR is false, but HAS-DIL is true, the status bar appears, but is only used as dynamic information line.

When you use Natural code, the above step may look like this:

Example:

```
...
PERSDATA-DIALOG.HAS-STATUS-BAR := TRUE  /* Set HAS-STATUS-BAR To TRUE
PERSADTA-DIALOG.STATUS-TEXT := 'HELLO'  /* Set the text to 'Hello'
...
```

Note: The STATUS-TEXT and the DIL-TEXT are displayed in the same area if the dialog has a status line and a text is displayed on the DIL.

Working with Status Bar Controls

Note:
Status bar controls are not to be confused with the traditional dialog status bar which is created by selecting the 'status bar' check box in the Dialog Atttributes window in the Dialog Editor, or by setting the dialog's HAS-STATUS-BAR attribute at run-time. If you are using status bar controls, you should leave the 'status bar' checkbox unchecked and not set the HAS-STATUS-BAR attribute.

Creating a Status Bar Control

Status bar controls are created in the dialog editor in the same way as other standard controls (such as list boxes or push buttons) are. That is, they are either created statically in the Dialog Editor via the Insert menu or by drag and drop from the Insert tool bar, or dynamically at run-time by using a PROCESS GUI ACTION ADD statement with the TYPE attribute set to STATUSBARCTRL.

Unlike most other control types, status bar controls cannot be nested within another control and cannot be created within an MDI child dialog. In an MDI application, the status bar control(s) must belong to the MDI frame dialog.

A status bar control may have zero or more panes associated with it. Panes may be defined in the Dialog Editor from within the status bar control's attribute window, or at run-time by performing a PROCESS GUI ACTION ADD statement with the TYPE attribute set to STATUSBARPANE.

Using status bar controls without panes

A status bar control without panes offers restricted functionality, because most attributes providing access to the enhanced functionality of status bar controls are only supported for status bar panes. If you wish to do more with a status bar control than simply display a line of text, but don’t need to split up the status bar control into multiple sections, you should create a single pane that occupies the full width of the status bar control.

Stretchy vs. non-stretchy panes

If panes are defined for a status bar control, it should be decided whether each pane should stretch (or contract) when the containing dialog is resized, or whether it should maintain a constant width. The former are referred to here as ’stretchy’ panes, and the latter as ’non-stretchy’ panes.

There is no explicit flag in the Status Bar Control Attributes window to mark a pane as stretchy or non-stretchy. Instead, any pane defined with a width (RECTANGLE-W attribute) of 0 is implicitly assumed to be a stretchy pane, whereas any panes with a non-zero width definition are implicitly assumed to be fixed-width panes of the specified width (in pixels). Because the RECTANGLE-W attribute defaults to 0, all panes are initially stetchy when defined in the Dialog Editor.

The width of a visible stretchy pane is determined by taking the total width available for all panes in the status bar control, subtracting the widths of all visible fixed-width panes, then dividing the result by the number of visible stretchy panes.

Note:
The total available width for all panes normally excludes the sizing gripper, implying that the last pane stops short of the gripper, if present. However, if the status bar control has exactly one pane, and that pane is a stretchy pane, the full width of the dialog (including any sizing gripper) is used.

Outputting text to a status bar control

Text can be output to the status bar control in one of three ways:

1. For status bar controls with panes, by setting the STRING attribute of the pane whose text is to be set.
2. By setting the STRING attribute of the status bar control itself, which is equivalent to setting the STRING attribute of the first stretchy pane (if any) for status bar controls with panes.
3. By setting the STATUS-TEXT attribute of the dialog. This is equivalent to setting the STRING attribute of the status bar control (if any) identified by the dialog’s STATUS-HANDLE attribute.

Note that the last method is often the most convenient for setting the message text, because it does not require a knowledge of the status bar control or pane handles.

Example:

```
DEFINE DATA LOCAL
01 #DLG$WINDOW  HANDLE OF WINDOW
01 #STAT-1      HANDLE OF STATUSBARCTRL
01 #PANE-1      HANDLE OF STATUSBARPANE
END-DEFINE
...
#DLG$WINDOW.STATUS-HANDLE := #STAT-1
...
#PANE-1.STRING := ’Method 1’
...
#STAT-1.STRING := ’Method 2’
...
#DLG$WINDOW.STATUS-TEXT := ’Method 3’
```

Note:
The Dialog Editor automatically generates code to set the STATUS-HANDLE attribute to the first status bar control (if any). Therefore, the STATUS-HANDLE attribute only needs to be set explicitly if you are dynamically creating status bar controls, or if you have defined more than one status bar control in a dialog, and wish to switch between them.

Sharing a status bar in an MDI applications

Because status bar controls cannot be created for MDI child dialogs, it is convenient to not have to define multiple status bar controls in the MDI frame dialog. An alternative method is to define just a single status bar, and share it between each child dialog. This can be achieved as follows:

1. Define all possible panes you wish to use in your application within a single status bar control in the MDI frame dialog.
2. Mark all panes as ’shared’.
3. Export the handles of all panes to corresponding shadow variables in a GDA, so that the MDI child dialogs can access them directly.
4. In the COMMAND-STATUS event handler, set the VISIBLE attribute of all panes you wish to display for that dialog to TRUE. All other panes will be automatically made invisible.

Note:
In the COMMAND-STATUS event, you must also set the ENABLED state of any commands (signals, or menu or tool bar items which do not reference another object via their SAME-AS attribute) associated with the dialog, otherwise they will be automatically disabled. The commands associated with the dialog are all non-shared commands for the MDI frame and all shared commands for the active MDI child (or MDI frame, if no MDI child dialog is active).

Pane-specific context menus

Context menus are defined for the status bar control and not per-pane. However, if you wish to ensure that the context menu for a status bar control only appears when the user right clicks a particular pane, you can associate a context menu with the status bar control, but suppress it if the user clicks outside that pane.

Example:

```
DEFINE DATA LOCAL
01 #CTXMENU-1   HANDLE OF CONTEXTMENU
01 #STAT-1      HANDLE OF STATUSBARCTRL
01 #PANE-1      HANDLE OF STATUSBARPANE
01 #PANE-2      HANDLE OF STATUSBARPANE
01 #PANE-3      HANDLE OF STATUSBARPANE
01 #PANE        HANDLE OF STATUSBARPANE
01 #X (I4)
01 #Y (I4)
END-DEFINE
...
#STAT-1.CONTEXT-MENU := #CTXMENU-1
...
DECIDE ON FIRST *CONTROL
   ...
   VALUE #CTXMENU-1
       DECIDE ON FIRST *EVENT
        ...
    VALUE 'BEFORE-OPEN'
    /* Get click position relative to status bar control
    PROCESS GUI ACTION INQ-CLICKPOSITION WITH
      #STAT-1 #X #Y GIVING *ERROR
       /* Get pane (if any) at specified position
       PROCESS GUI ACTION INQ-ITEM-BY-POSITION WITH
      #STAT-1 #X #Y #PANE
    /* Only show context menu if user clicked in second pane
       IF #PANE = #PANE-2
    #CTXMENU-1.ENABLED := TRUE
       ELSE
    #CTXMENU-1.ENABLED := FALSE
    END-IF
      ...
 END-DECIDE
 ...
END-DECIDE
...
END
```

Note:
If you wish to display a different context menu for different status bar panes, the menu items must be created dynamically in the context menu's BEFORE-OPEN Event.

Working with Dynamic Information Line and Status Bar

When you are working with both a Dynamic Information Line (DIL) and a status bar, the combination of the HAS-DIL and HAS-STATUS-BAR attributes determines whether and when DIL-TEXT and STATUS-TEXT values will be displayed:

HAS-DIL	HAS-STATUS-BAR	DIL-TEXT	STATUS-TEXT
TRUE	TRUE	displayed	displayed
TRUE	FALSE	-	-
FALSE	TRUE	-	displayed
FALSE	FALSE	-	-

If HAS-DIL and HAS-STATUS-BAR are TRUE, the DIL-TEXT will overlap the STATUS-TEXT value and vice versa, depending on which was modified last.

Adding a Maximize/Minimize/System Button

To add a Maximize/Minimize/System button to your dialog

Open the "Dialog Attributes" window. Check the "System Button" or "Maximizable"or "Minimizable" entry.

When the "System Button" entry is checked, the dialog’s standard control menu is available. This includes the control-menu box (to close the dialog), the title bar, and the Maximize and Minimize buttons.

Defining Color

You can define colors for dialogs and dialog elements. These can be foreground colors and background colors. To do this, you use the following attributes:

- BACKGROUND-COLOUR-NAME
- BACKGROUND-COLOUR-VALUE
- FOREGROUND-COLOUR-NAME
- FOREGROUND-COLOUR-VALUE

You can assign only standard colors to the attributes ending with NAME. The attributes ending on VALUE, however, can be assigned customized colors following the RGB model.

You can set colors:

- in an attributes window, or
- in event-handler code.

You can directly assign a value to the attributes ending with NAME. If you want to assign a value to an attribute ending with VALUE, you must set the NAME attribute to the value CUSTOM. If you do not set the NAME attribute to the value CUSTOM, the VALUE attribute is ignored.

Examples:

```
#DIA.BACKGROUND-COLOUR-NAME:= MAGENTA     /* Assign a value to a NAME
                                          /* attribute
#DIA.BACKGROUND-COLOUR-NAME:= CUSTOM      /* Set NAME to CUSTOM
#DIA.BACKGROUND-COLOUR-VALUE:= H’FF0000’  /* Then assign Red, Green, and
                                          /* Blue values to the VALUE
                                          /* attribute (hexadecimal)
```

Note: You can not use all customized colors in all parts of the user interface. Colors in text, for example, must always be monochrome.

When setting a color in an attributes window, you have three possibilities:

- Use the attribute ending with NAME and leave the value at DEFAULT. You can also do this in code. Your color will then be determined by your color settings in the windowing system.
- Use the attribute ending with NAME by pulling down the list-box and choose one of the predefined colors.
- Define your own color by using the attribute ending with VALUE.

To define a color

1. Click on the "Custom" push-button control right of the "Background color" entry.
   A dialog box appears.
2. Select one of the predefined colors or click on the "Define Custom Colors" push-button control. To set the red, green, and blue values, use the cursor to select the desired color or enter a value from 1 to 253 in the red, green,

and blue value display fields.

3. Click on the "Add to Custom Color" push-button control. To save the newly defined color, click on the OK button in the dialog box.
   The newly defined color is now selected by default.
4. To set it, close the attributes window.

Adding Text in a Certain Font

► To choose a specific font for the text assigned to a dialog element (for example, the caption on a push-button control), you use the dialog element's attributes window

1. Click on the "..." push-button control to the right of the "Font" entry.
   A dialog box opens.
2. From the list of available fonts, select a font type, for example "Times New Roman".
3. From the list of styles available for the font type, select a font style, for example *"italics"*.
4. From the list of sizes available for the font type and style, select a font size, for example "10".
   A sample of your selected font will be displayed.
5. To set it: Close the attributes window.

Note: When adding centered or right-aligned text in a dialog element, the following minimum heights of the dialog element apply (RECTANGLE-H attribute): 4-point font - height of 8; 8-point - 22; 12-point - 24.

Additionally, the dialog editor allows selecting a font for the whole dialog in the dialog attributes window. This font is defined in the FONT-STRING attribute and is valid for the dialog and each of its children. A major advantage of selecting a font for the whole dialog is that if the chosen font is too large or too small for the dialog layout, you change the FONT-STRING attribute once instead of going through all children of the dialog.

Initially, the FONT-STRING attribute must be set as a parameter while the dialog is being created with PROCESS GUI action ADD. If a dialog element inside the dialog contains text with no particular font assigned to it, this text will be displayed in the font specified by FONT-STRING.

For more information on the FONT-STRING attribute and the way its value must be specified, see the Natural Dialog Components Manual.

Adding Online Help

From an application written with the dialog editor, you can invoke help for a specified help topic ID. Please bear in mind that you will have to create parts of the help associated with these help topic IDs outside the Natural development environment. You will also have to compile the help with the platform-specific help compiler.

To keep an overview of all the different help sections in an application, Natural provides you with the help organizer. With this organizer,

- you assign a help ID (HELP-ID attribute value) to a specific dialog element;
- you write the help text for the associated help topic; this text is converted to a .rtf file to be processed by the help compiler;
- you optionally define the help topic’s keywords;
- you optionally assign a help compiler macro to the help topic;
- and optionally you add a comment for your internal documentation purposes.

To create a help topic

1. Invoke the help organizer’s main dialog.
2. Select a particular dialog element.
3. Generate a new help topic ID.
4. Return to the help organizer main dialog.
5. Assign the generated help topic ID.
6. Enter the external definitions for the help topic ID, such as the help topic text and the topic name.
7. Return to the help organizer main dialog.
8. Go to the topic list and see whether this new help topic fits your general organization of the help file to be created.
9. Return to the help organizer main dialog.
10. Save everything.

A dialog or dialog element can also be assigned a HELP-ID number independently of the help organizer.

To do so

Open the corresponding attributes window. Enter a numeric value in the "Help ID" entry.

You must use the help topic’s .h file to map the numerical ID that you enter here to the corresponding help topic ID (created by a markup in the .hlp file).

Natural expects the help file to be located in the resource (RES) subdirectory of the current library or one of the STEPLIBs, or in the directory referred to by the environment variable NATGUI_BMP. By default, Natural searches for a help file with the same name as the current library, but you can explicitely set the name of the help file via the HELP-FILENAME attribute.
If no file extension is specified, Natural searches for a compiled HTML help file with the extension ".chm" first, then (if not found) for a WinHelp help file with the extension ".hlp".
Thus, if no file extension is specified, it is possible to upgrade from using WinHelp to using HTML help without changing the Natural program. Note, however, that the Help Organizer only supports WinHelp. If you wish to create HTML help content, you must use an external help authoring tool to do so.

Whenever an end user presses F1 in an active dialog, Natural first queries for a file with the value of the HELP-FILENAME attribute plus the extension ".h*nn*" where *nn* is the Natural language code. If it does not find such a file, it queries for a file with the value of HELP-FILENAME plus the extension ".hlp"

Whenever the dialog element has the focus and the end user presses F1, Natural jumps to this help ID.

Note: When adding online help to an application, it is recommended to assign a HELP-ID number to each dialog and to write help texts for the dialogs. If the end user selects a dialog element to which no HELP-ID was assigned and presses F1 to request help, help on the current dialog will come up. If no HELP-ID was assigned to a dialog element, Natural will check whether the dialog element’s parent - the dialog - has a HELP-ID. If not, Natural will check whether the dialog’s parent - the dialog one level higher - has a HELP-ID, and so on, until the top-level dialog is reached.

To build a help file

1. Go to your command promt.
2. Change to the directory referred to by the environment variable $NATGUI_BMP.
3. Issue the command "HCRTF -X *helpfilename*".

Note:
This assures that the directory containing HCRTF.EXE is specified in the PATH environment variable.

To test a help file

1. Invoke a dialog in your application.
2. Press F1.

The help topic for the dialog should appear.

Alternatively, the help file can be conveniently built and tested interactively by opening the .hpj file in the Help Compiler Workshop (HCW.EXE).

To display help in a popup window

1. Check the Popup Help option in the dialog attributes window.
2. Run the dialog.
3. Press F1 with the focus on a control which has a help ID associated with it.

The help topic associated with the focus control should appear in a popup window.

Defining Mnemonic and Accelerator Keys

There are two ways of providing keyboard commands:

- A mnemonic key is determined by an underlined character in a visible dialog element, for example a menu item. The end user can select the menu item by pressing ALT+mnemonic key, for example ALT+A.
- An accelerator key is defined in the ACCELERATOR attribute. By pressing this key, the end user causes a double-click or click event for the dialog element regardless of whether the dialog element is visible or not, as long as the dialog element is enabled.

Defining a Mnemonic Key

You define a mnemonic key in the dialog element’s STRING attribute by specifying "&" before the desired character. At runtime, the character will be underlined. Example: the STRING attribute value "E&xplanation" will be displayed as "Explanation" at runtime.

If you define a mnemonic key with a text constant control or a group frame control, and the end user presses the mnemonic key at runtime, the next dialog element in the control sequence will get the focus. For example, if the next dialog element after a text constant control is an input-field control, the text constant control’s mnemonic key sets the focus to the input-field control. Whenever you disable such an input-field control at runtime, you should also disable the corresponding text constant control.

You can define mnemonic keys in the STRING attribute of the following types of dialog elements: group frame control, menu item, push-button control, radio-button control, text-constant control, toggle-button control, toolbar item.

You can still display an "&" in your runtime STRING by specifying "&&". Example: "A&&B" will be displayed as "A&B".

Note:
In recent Windows versions (e.g. Windows 2000), mnemonic characters are, by default, not underlined until the <Alt> key is pressed. However, this new behavior can be disabled by the user, such that mnemonic characters are always underlined. For example, this can be achieved on the English version of Windows 2000 by unchecking the

"Hide Keyboard navigation indicators until I use the Alt key"

option under:

"Start/Control Panel/Display/Effects."

Defining an Accelerator Key

You define an accelerator key by setting the ACCELERATOR attribute to a key or a key combination for the dialog element, for example to "F6" or "CTRL+1". If the end user presses the accelerator key, the double-click event occurs for the dialog element, or if no double-click event is available, the click event occurs. The accelerator key does not work if the corresponding event is suppressed, or if the dialog element is disabled.

Standard system accelerators such as "Alt+Esc", "Ctrl+Esc", "Alt+Tab" and "Ctrl+Alt+Del" can be defined as accelerators, but do not cause the dialog element’s click or double-click event to be triggered. Instead, they cause the associated system functionality to be invoked. The same applies to standard MDI accelerators (such as "Ctrl+F4" and "Ctrl+F6") if used within MDI applications and to any accelerators belonging to in-place activated servers (e.g. ActiveX controls which currently have the focus).

Note that user-defined accelerator keys overwrite identical user-defined shortcut keys associated with desktop items.

If the same accelerator key is associated with more than one dialog element, the dialog element whose click or double-click event is triggered is not defined.

A dialog element which references another via its SAME-AS attribute inherits the accelerator of the referenced object. For example, if a menu item references a signal, and the signal's accelerator is "Ctrl+Alt+X", then querying the menu item's ACCELERATOR attribute will also return "Ctrl+Alt+X". However, the accelerator, if pressed, will only trigger a click event for the referenced dialog element (i.e., the signal in this example).

Accelerators of the form "Alt+X", where "X" is one of the alphabetic characters, should be avoided, because they are "reserved" for use as keyboard mnemonics.

Displaying Accelerator Keys in Menus

In order to show accelerators for menu items, the menu text needs to first be appended with a tab (h'09') character and then appended with the text for the accelerator. This cannot be done statically in the dialog editor's menu editor, because there is no way to enter a tab character into the string definition. However, the accelerators may be appended dynamically using a generic piece of code which iterates round all menu items for a dialog. This is illustrated by the following external subroutine, which can conveniently be called from within a dialog's AFTER-OPEN event.

Example:

```
    DEFINE DATA
    PARAMETER
      1 #DLG$WINDOW  HANDLE OF WINDOW
    LOCAL
      1 #CONTROL     HANDLE OF GUI
      1 #COMMAND     HANDLE OF GUI
    LOCAL USING NGULKEY1
    END-DEFINE
    *
    DEFINE SUBROUTINE APPEND-ACCELERATORS
    #CONTROL := #DLG$WINDOW.FIRST-CHILD
    REPEAT UNTIL #CONTROL = NULL-HANDLE
      IF #CONTROL.TYPE = SUBMENU OR #CONTROL.TYPE = CONTEXTMENU
     #COMMAND := #CONTROL.FIRST-CHILD
     REPEAT UNTIL #COMMAND = NULL-HANDLE
    IF #COMMAND.ACCELERATOR <> ' '
        COMPRESS #COMMAND.STRING H'09' #COMMAND.ACCELERATOR INTO
    #COMMAND.STRING LEAVING NO SPACE
    END-IF
       #COMMAND := #COMMAND.SUCCESSOR
     END-REPEAT
     END-IF
     #CONTROL := #CONTROL.SUCCESSOR
   END-REPEAT
   END-SUBROUTINE
   END
```

This dynamic technique has the advantage that the accelerator does not, in effect, have to be defined twice (i.e., for the ACCELERATOR and STRING attributes of the menu item).

Note that if the target language is not English, the ACCELERATOR attribute value will probably have to be translated before being appended to the menu item string.

Dynamic Data Exchange - DDE

Concepts

DDE is a protocol defined by Microsoft Corp. to enable different applications to exchange data. This means that, for example, an application written in Natural may exchange data with a spreadsheet, because they are both able to process the DDE protocol. An application that processes the DDE protocol communicates with another DDE application via standardized messages. One of the applications is defined as the client, the other as the server. Client and server are holding a DDE conversation.

Note: For an overview of DDE concepts and terminology, see your Microsoft Windows documentation.

Data in a DDE conversation is identified by a three-level hierarchy:

- service,
- topic,
- item.

A DDE conversation is established whenever a client requests a *service* from a DDE server. A DDE server offers one or more *services* to all active applications.

For each service, a DDE server may offer any number of *topics*. The DDE client then requests a conversation on a *topic* of a *service*.

In a conversation on a *topic* of a *service*, the DDE client and the DDE server uniquely identify data to be exchanged by an *item* name.

A DDE server may support a number of services, which in turn may consist of a number of topics, which themselves may contain a number of items.

With Natural, you can develop both DDE client applications as well as DDE server applications. You may, for example, write a Natural DDE client application that requests data from a spreadsheet acting as a DDE server, or you may write a Natural DDE server application that supplies a word processor (DDE client) with data.

To develop DDE client and DDE server applications, the following functionality is provided:

- A number of NGU-prefixed subprograms in library SYSTEM; these send messages and data as defined in the parameter data area "NGULDDE1"
- a parameter data area (NGULDDE1) which describes the parameters used by the subprograms in a DDE conversation (the "DDE-VIEW");
- a DDE-Client event and a DDE-Server event which handle DDE messages.

You develop a DDE server application by reacting to the DDE-Server event and by using the NGU-SERVER-prefixed subprograms from library SYSTEM to register services and topics and to send messages and data to the DDE client application.

You develop a DDE client application by reacting to the DDE-Client event and by using the NGU-CLIENT-prefixed subprograms from library SYSTEM to initiate conversations and send requests and other DDE commands to DDE server applications.

You always have to include the parameter data area NGULDDE1 and the local data area NGULFCT1 in your client or server dialog. (You need NGULFCT1 in order to use the NGU-prefixed subprograms in library SYSTEM).

Developing a DDE Server Application

Registering/Unregistering Services and Topics

Before a DDE server application can be addressed by a DDE client application, it must register its service names and all supported topics for the services. You use subprogram NGU-SERVER-REGISTER to do this for each service/topic the DDE server supports. Registering will usually be handled in the "after open" event of the base dialog.

When registering a service/topic for the first time, you will need to supply Natural with the dialog-ID of the dialog that will function as the server and that will therefore receive all DDE messages from clients. This is done by setting the DDE-VIEW.CONV-ID to the respective dialog-ID and also by setting DDE-VIEW.MESSAGE to the string ’DLGID’.

Note that at a later time you are able to add more topics to a service or even entirely new services. You can also make a topic unavailable by using subprogram NGU-SERVER-UNREGISTER.

Getting Data From The Client

After successful registration, it is possible that the DDE server application receives DDE messages from a DDE client application which is establishing a conversation on a registered topic of a service.

Such messages for a DDE server are received in the DDE-Server event of the dialog. At the beginning of the event-handler section, it is necessary to fill the DDE-VIEW with the client’s message data. This is done by using subprogram NGU-SERVER-GET-DATA. After reading the data, it will be necessary to act based on the client message received. The possible messages and their meaning are explained in the description of subprogram NGU-SERVER-GET-DATA.

Sending Data To The Client

In many cases, the client message ultimately requires the server to send data to the client. This is achieved by using the subprogram NGU-SERVER-DATA.

Terminating DDE Server Operation

Whenever DDE server operation is supposed to terminate, you use the subprogram NGU-SERVER-STOP. It unregisters all services and terminates all active conversations. You terminate the server application with the CLOSE DIALOG statement.

Developing a DDE Client Application

Connecting With The DDE Server Application

In order to establish a conversation with a DDE server application, a DDE client application must call the subprogram NGU-CLIENT-CONNECT with the service and topic name of the server it wants to connect. In order to receive the appropriate DDE events from a server, it is necessary to set the DDE-VIEW.CONV-ID to the client’s dialog-ID and also to set DDE-VIEW.MESSAGE to the string ’DLGID’. The call will return a unique conversation ID in DDE-VIEW.CONV-ID. This value must be set appropriately in all further communication with the server.

Using The Services of a DDE Server Application

The client has several options to use the services of a server once a conversation has been established. It can

- request data on a specific item (using NGU-CLIENT-REQUEST),
- send data to the server (using NGU-CLIENT-POKE),
- ask the server to execute a command (using NGU-CLIENT-EXECUTE), or
- establish a warm or hot link to the server (using NGU-CLIENT-ADVISE-HOT,

NGU-CLIENT-ADVISE-WARM and NGU-CLIENT-ADVISE-TERM).

Receiving Data From The DDE Server Application

The DDE client will receive data or other messages from the DDE server via the client dialog's DDE-Client event. Whenever a server has sent a message, this event occurs. The message contents must first be retrieved using NGU-CLIENT-GET-DATA. This will fill the DDE-VIEW structure appropriately. The client must then determine which message (DDE-VIEW.MESSAGE) has arrived and react appropriately. The possible messages are listed in the description of subprogram NGU-CLIENT-GET-DATA.

Disconnecting From The DDE Server Application

Whenever the client determines that the conversation is no longer needed, a call to NGU-CLIENT-DISCONNECT must be issued to inform the server that the conversation is to be terminated.

Terminating DDE Client Operation

Whenever the client application terminates or wants to stop using DDE, it needs to call NGU-CLIENT-STOP. This informs Natural to close all active conversations of the client and shut down DDE operation for the application.

Return Codes

Possible return codes are described in this section:

Note: Each error-code description is not necessarily comprehensive. In these cases, the description is marked with an asterisk (*).

Code	Meaning
-1	You have specified an incorrect command or command parameter. Ensure that your DDE data area is of the correct type and that the command is correct.
0	The function was processed correctly.
1	This value is returned when an application has attempted to initialize with the DDEML library more than once. Check the logic of your program. Also ensure that the DDEML was exited correctly during the last run of the program.
2	This value may be returned from the server-initialize function if you have run the program before and not exited the DDEML correctly. It is also returned by a call-back function, whenever the requested service failed.
	An error occurred in the underlying layer.*
3	The conversation ID referenced does not represent an active conversation. Check if you have specified a correct service name.
4	The application could not initialize with the DDE library as the maximum number of instances are connected.
5	The DDEML communication has not been initialized. You must initialize with the DDEML before any DDE activity can take place.
6	Memory allocation problems encountered. This error might occur if the queue of messages for either part in the conversation becomes too long. *
7	A service, topic or item name was longer than 255 characters. Check if your fields are correctly specified for DDE-VIEW and make sure that you are not attempting to place a string longer than 255 characters in any one of the above variables.
8	An error occurred in the DDE library. Contact SOFTWARE AG Support.*

Code	Meaning
9	Parameters passed to this function were illegal. This can be returned by any function call. Check your parameters.
10	"Server Type Link" is supported but no call-back function for UNLINK is passed to the function "PIDsRegisterTopic". *
11	An attempt was made to remove a topic for which at least one conversation is still active. This includes trying to unregister a topic for which a conversation still exists.
12	The service/topic referenced has not been registered with the function "PIDsRegisterTopic".
13	No links were active for the DDE-VIEW.SERVICE when the NGU-Server-Data subprogram was used. Check your service name and use the DDE-SPY in the SDK Tool Kit to see what services are available.
14	The requested type of link is invalid.
15	The transaction ID is corrupted. Check the value of your transaction ID in your DDE view.
16	The client application requested a conversation and prior to that, no function was specified to send the data for the links.
17	An asynchronous transaction was requested, but the client application did not specify a function to send details of the completed transaction. Such a function must be specified when the conversation is initialized.
18	A synchronous transaction timeout expired. The amount of time taken for your transaction to complete was longer than the TIMEOUT value in your DDE-VIEW structure. Increase the TIMEOUT value or set it to "-1" for indefinite waiting.
19 - 24	For internal use only.

Object Linking and Embedding - OLE

What Is OLE In The Natural Context?

Natural supports the following OLE technologies:

- OLE Documents
- OLE Visual Editing (In-place Activation)
- ActiveX Controls

If you are new to OLE, it is highly recommended that you first get a basic overview by referring to one of the various sources available. One such source, for example, is the Microsoft Win32 software development kit documentation.

OLE Documents Support

OLE documents is a technology that integrates different Windows applications seamlessly so that the end user can concentrate on the data rather than on handling the different applications. With OLE you can, for example, embed a Word for Windows document in a Natural dialog. Whenever the end user enters the text container to edit the document, the entire Word functionality is available. Thus, the end user does not have to invoke Word.

OLE Documents Support is provided by the Natural dialog element OLE container control. For more information, see the section The OLE Container Control.

The OLE documents technology defines container and server applications. A container application is an application that is able to use objects created by a server application. These objects are used by linking or embedding them. In this context, Natural is the container application because the dialog editor provides an OLE container control. A typical server application is Microsoft Word; the Word documents would then be the objects used by Natural.

Embedding and Linking

- Linking means that the content of a document is accessed via a link to an external file. This file is stored in the server’s format (for example, a file in ".rtf" format would be stored in a file system outside Natural; the server residing in this external file system would be Microsoft Word).
- Embedding means that the content of a document is maintained in the container application and is stored in the container’s internal format. Embedded documents are created
  - either by building them from scratch in the container application;
  - or by loading an external document.

Embedded objects are edited by visual editing ("in-place activation"), whereas linked objects must be opened in an extra server window for editing.

Natural provides the dialog element OLE container control for embedding and linking documents. Furthermore, Natural provides actions to save and load embedded documents in internal Natural format. By default, these embedded objects in internal format are stored and retrieved in the %NATGUI_BMP% directory with a default extension of ".neo" (Natural Embedded Object).

If you want the OLE container control to display an embedded object when the dialog starts

1. Invoke the container control’s attribute window.
2. Set the Type entry to "Existing OLE Object".
3. Select a file specification in the Name field.

If you want to display an embedded object dynamically at runtime

Use the PROCESS GUI statement action OLE-READ-FROM-FILE (see also Dialog Components Manual, Chapter Executing Standardized Procedures).

If you want the OLE container control to display a linked object when the dialog starts

1. Invoke the container control's attribute window.
2. Set the Type entry to "OLE Server".
3. In the "Select OLE Server or Document" dialog that comes up, select "Create From File" and select a file specification.

If you want to display a linked object dynamically at runtime

Assign the file specification of the external document to the attribute SERVER-OBJECT of the OLE container control (see also Dialog Components Manual, Chapter Attributes).

Visual Editing - In-place Activation

In-place activation means that the end user is able to activate a server application in the container application's window. Such a server application is related to an object embedded in a Natural dialog's OLE container control. The server application is activated by double-clicking on the OLE container control. The Natural dialog's toolbar and menu-bar control are then merged with the server application's menu and toolbar. The dialog now contains toolbar items and menu items that enable you to edit the object with the help of the server's functionality.

ActiveX Controls Support

ActiveX controls support enables the Natural programmer to use the many third-party ActiveX controls inside a Natural dialog. Natural enables you to access the ActiveX controls properties and methods direct and to program the ActiveX controls events.

ActiveX controls support is provided by the Natural dialog element "ActiveX control". For more information, see Working with ActiveX Controls.

The OLE Container Control

Creating an OLE Container Control

You can create an OLE container control either statically in the dialog editor or dynamically at runtime.

Creating an OLE Container Control in the Dialog Editor

The OLE container control enables you to integrate server applications. You can integrate server applications in the following three ways, as indicated by the "Object Information" group frame, "Type" entry of the OLE container control's attributes window.

- Type: New OLE object. You create an OLE container control that acts as a placeholder for the insertable object. At runtime, your end user can create the embedded object by starting the server application. The embedded object can then be saved as Natural embedded object (.neo file).
- Type: Existing OLE object. Your end user changes an existing embedded object in the OLE container control. The embedded object is saved as Natural embedded object (.neo file).
- Type: OLE server. You create a native OLE object in your application or you create a link to an external object.

To create an OLE container control in the dialog editor

1. In the dialog editor main menu, choose "Insert", then "OLE Container".
2. Draw a rectangle by holding down the right mouse button, dragging the mouse vertically/horizontally and releasing the mouse button.

An empty OLE container is created.

To display a document in the OLE container when starting the dialog

1. Double-click the OLE container control to invoke the attribute window.
2. In the "Type" selection box, choose "OLE server" for linking an external document.
   Or choose "Existing OLE object" for reading in an embedded object.
3. Press the "..." button to select the external or embedded object file.

Creating an OLE Container Dynamically At Runtime

Before you enter the examples in an event-handler section, declare a handle variable for the OLE container control in the local data area of the dialog:

```
01 #OCT-1 HANDLE OF OLECONTAINER
```

Example for creating an OLE container control at runtime and linking an external document:

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
   HANDLE-VARIABLE = #OCT-1
   TYPE = OLECONTAINER
   SERVER-OBJECT = 'PICTURE.BMP'
   RECTANGLE-X = 56
   RECTANGLE-Y = 32
   RECTANGLE-W = 336
   RECTANGLE-H = 160
   PARENT = #DLG$WINDOW
   SUPPRESS-CLICK-EVENT = SUPPRESSED
   SUPPRESS-DBL-CLICK-EVENT = SUPPRESSED
   SUPPRESS-CLOSE-EVENT = SUPPRESSED
   SUPPRESS-ACTIVATE-EVENT = SUPPRESSED
   SUPPRESS-CHANGE-EVENT = SUPPRESSED
END-PARAMETERS GIVING *ERROR
```

Example for creating an OLE container control at runtime and embedding a Natural embedded object:

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
   HANDLE-VARIABLE = #OCT-1
   TYPE = OLECONTAINER
   EMBEDDED-OBJECT = 'SLIDE.NEO'
   RECTANGLE-X = 56
   RECTANGLE-Y = 32
   RECTANGLE-W = 336
   RECTANGLE-H = 160
   PARENT = #DLG$WINDOW
   SUPPRESS-CLICK-EVENT = SUPPRESSED
   SUPPRESS-DBL-CLICK-EVENT = SUPPRESSED
   SUPPRESS-CLOSE-EVENT = SUPPRESSED
   SUPPRESS-ACTIVATE-EVENT = SUPPRESSED
   SUPPRESS-CHANGE-EVENT = SUPPRESSED
   END-PARAMETERS GIVING *ERROR
```

Clearing or Deleting an OLE Container At Runtime

This section contains examples for clearing and deleting an OLE container at runtime.

Before you enter the examples in an event-handler section, declare a handle variable for the OLE container control in the local data area of the dialog:

```
01 #OCT-1 HANDLE OF OLECONTAINER
```

Example for clearing (removing the document of) the OLE container control:

```

PROCESS GUI ACTION CLEAR WITH #OCT-1
```

Example for deleting the OLE container control:

```
PROCESS GUI ACTION DELETE WITH #OCT-1
```

OLE Container Controls And The Dialog’s Menu Bar

The menu item attribute MENU-ITEM-OLE can have four different values which detemine if and where the menu item in question is displayed during in-place activation of a server (see also Dialog Components Manual, Chapter Attributes).

The menu item attribute MENU-ITEM-TYPE also has the value MT-OBJECTVERBS. This enables you to have the OLE container control display the available server actions (command verbs) in this menu item (see also Dialog Components Manual, Chapter Attributes).

Other OLE Container Control Functionality

While a document is displayed in an OLE container control, the end user has the possibility to activate the default command verb of the server by double-clicking inside the OLE container control’s rectangle. This is equivalent to executing the PROCESS GUI statement action OLE-ACTIVATE (see also Dialog Components Manual, Chapter Executing Standardized Procedures). Furthermore, the end user can select a server command verb by displaying a popup menu. You display this popup menu by holding down the right mouse button inside the OLE container. Then you select the desired command verb and release the mouse button.

If the MODIFIABLE attribute of an OLE container control is set to FALSE, a double-click on the container does not start the default command verb of the server and holding down the right mouse button does not show the popup menu with the available server command verbs (see also Dialog Components Manual, Chapter Executing

Standardized Procedures).

During visual editing (in-place activation), the server uses the Natural dialog for the editing of the document. The server does its work as a task on its own and the Natural processing continues. Thus, it is possible to execute event code and, for example, to limit the visual editing to a certain time by specifying PROCESS GUI ACTION OLE-DEACTIVATE, WITH #OCT-1 in a timer's event section (see also Dialog Components Manual, Chapter Executing Standardized Procedures).

Attributes, Events and PROCESS GUI Statement Actions

The following sections list all the attributes, events and PROCESS GUI statement actions that apply specifically to the OLE container control.

Attributes

The OLE-specific attributes provided with the OLE container control are:

- EMBEDDED-OBJECT (see Dialog Components Manual, Chapter Attributes)
- ICONIZED (see Dialog Components Manual, Chapter Attributes)
- OBJECT-SIZE (see Dialog Components Manual, Chapter Attributes)
- SERVER-OBJECT (see Dialog Components Manual, Chapter Attributes)
- SERVER-PROGID (see Dialog Components Manual, Chapter Attributes)
- SUPPRESS-ACTIVATE-EVENT (see Dialog Components Manual, Chapter Attributes)
- SUPPRESS-CLOSE-EVENT (see Dialog Components Manual, Chapter Attributes)
- ZOOM-FACTOR (see Dialog Components Manual, Chapter Attributes)

Event

This OLE-specific event occurs when a server application is activated:

- Activate event (see Dialog Components Manual, Chapter Events)

PROCESS GUI Statement Actions

The OLE-specific PROCESS GUI statement actions provided with the OLE container control are:

- OLE-ACTIVATE (see Dialog Components Manual, Chapter Executing Standardized Procedures)
- OLE-DEACTIVATE (see Dialog Components Manual, Chapter Executing Standardized Procedures)
- OLE-GET-DATA (see Dialog Components Manual, Chapter Executing Standardized Procedures)
- OLE-INSERT-OBJECT (see Dialog Components Manual, Chapter Executing Standardized Procedures)
- OLE-READ-FROM-FILE (see Dialog Components Manual, Chapter Executing Standardized Procedures)
- OLE-SAVE-TO-FILE (see Dialog Components Manual, Chapter Executing Standardized Procedures)
- OLE-SET-DATA (see Dialog Components Manual, Chapter Executing Standardized Procedures)